

# Formalization of a Monitoring Algorithm for Metric First-Order Temporal Logic

Joshua Schneider      Dmitriy Traytel

February 6, 2026

## Abstract

A monitor is a runtime verification tool that solves the following problem: Given a stream of time-stamped events and a policy formulated in a specification language, decide whether the policy is satisfied at every point in the stream. We verify the correctness of an executable monitor for specifications given as formulas in metric first-order temporal logic (MFOTL) [1], an expressive extension of linear temporal logic with real-time constraints and first-order quantification. The verified monitor implements a simplified variant of the algorithm used in the efficient MonPoly monitoring tool [2]. The formalization is presented in a RV 2019 paper [4], which also compares the output of the verified monitor to that of other monitoring tools on randomly generated inputs. This case study revealed several errors in the optimized but unverified tools.

## Contents

<b>1</b>	<b>Traces and trace prefixes</b>	<b>2</b>
1.1	Infinite traces . . . . .	2
1.2	Finite trace prefixes . . . . .	4
<b>2</b>	<b>Finite tables</b>	<b>9</b>
<b>3</b>	<b>Abstract monitors and slicing</b>	<b>15</b>
3.1	First-order specifications . . . . .	15
3.2	Monitor function . . . . .	16
3.3	Slicing . . . . .	17
<b>4</b>	<b>Intervals</b>	<b>20</b>
<b>5</b>	<b>Metric first-order temporal logic</b>	<b>21</b>
5.1	Formulas and satisfiability . . . . .	21
5.2	Defined connectives . . . . .	25
5.3	Safe formulas . . . . .	25
5.4	Slicing traces . . . . .	26
<b>6</b>	<b>Monitor implementation</b>	<b>28</b>
6.1	Monitorable formulas . . . . .	28
6.2	The executable monitor . . . . .	29
6.3	Progress . . . . .	32
6.4	Specification . . . . .	36
6.5	Correctness . . . . .	37
6.5.1	Invariants . . . . .	37
6.5.2	Initialisation . . . . .	39
6.5.3	Evaluation . . . . .	40

6.5.4	Monitor step	53
6.5.5	Monitor function	53
6.6	Collected correctness results	55
<b>7</b>	<b>Slicing framework</b>	<b>56</b>
7.1	Abstract slicing	56
7.1.1	Definition 1	56
7.1.2	Definition 2	56
7.1.3	Definition 3	56
7.1.4	Lemma 1	56
7.2	Joint data slicer	56
7.2.1	Definition 4	57
7.2.2	Lemma 2	57
7.2.3	Theorem 1	57
7.2.4	Corollary 1	58
7.2.5	Definition 5	58
7.2.6	Theorem 2	58
7.2.7	Towards Theorem 3	59
7.2.8	Lemma 3	63
7.2.9	Definition of $J'$	63
7.2.10	Theorem 3	63

# 1 Traces and trace prefixes

## 1.1 Infinite traces

**coinductive** *ssorted* :: 'a :: linorder stream  $\Rightarrow$  bool **where**  
*shd*  $s \leq \text{shd } (stl\ s) \Longrightarrow \text{ssorted } (stl\ s) \Longrightarrow \text{ssorted } s$

**lemma** *ssorted\_siterate[simp]*:  $(\bigwedge n. n \leq f\ n) \Longrightarrow \text{ssorted } (\text{siterate } f\ n)$   
**by** (*coinduction arbitrary: n*) *auto*

**lemma** *ssortedD*:  $\text{ssorted } s \Longrightarrow s\ !!\ i \leq stl\ s\ !!\ i$   
**by** (*induct i arbitrary: s*) (*auto elim: ssorted.cases*)

**lemma** *ssorted\_sdrop*:  $\text{ssorted } s \Longrightarrow \text{ssorted } (\text{sdrop } i\ s)$   
**by** (*coinduction arbitrary: i s*) (*auto elim: ssorted.cases ssortedD*)

**lemma** *ssorted\_monoD*:  $\text{ssorted } s \Longrightarrow i \leq j \Longrightarrow s\ !!\ i \leq s\ !!\ j$

**proof** (*induct j - i arbitrary: j*)

**case** (*Suc x*)

**from** *Suc(1)[of j - 1] Suc(2-4) ssortedD[of s j - 1]*

**show** ?*case by (cases j) (auto simp: le\_Suc\_eq Suc\_diff\_le)*

**qed** *simp*

**lemma** *sorted\_stake*:  $\text{ssorted } s \Longrightarrow \text{sorted } (\text{stake } i\ s)$

**by** (*induct i arbitrary: s*)

(*auto elim: ssorted.cases simp: in\_set\_conv\_nth*

*intro!: ssorted\_monoD[of \_ 0, simplified, THEN order\_trans, OF \_ ssortedD]*)

**lemma** *sorted\_monoI*:  $\forall i\ j. i \leq j \longrightarrow s\ !!\ i \leq s\ !!\ j \Longrightarrow \text{ssorted } s$

**by** (*coinduction arbitrary: s*)

(*auto dest: spec2[of \_ Suc \_ Suc \_] spec2[of \_ 0 Suc 0]*)

**lemma** *sorted\_iff\_mono*:  $\text{ssorted } s \longleftrightarrow (\forall i\ j. i \leq j \longrightarrow s\ !!\ i \leq s\ !!\ j)$

**using** *ssorted\_monoI ssorted\_monoD by metis*

**lemma** *sorted\_iff\_le\_Suc*:  $\text{sorted } s \longleftrightarrow (\forall i. s !! i \leq s !! \text{Suc } i)$   
**using** *mono\_iff\_le\_Suc*[of *snth s*] **by** (*simp add: mono\_def sorted\_iff\_mono*)

**definition** *sincreasing*  $s = (\forall x. \exists i. x < s !! i)$

**lemma** *sincreasingI*:  $(\bigwedge x. \exists i. x < s !! i) \implies \text{sincreasing } s$   
**by** (*simp add: increasing\_def*)

**lemma** *sincreasing\_grD*:

**fixes**  $x :: 'a :: \text{semilattice\_sup}$

**assumes** *sincreasing*  $s$

**shows**  $\exists j > i. x < s !! j$

**proof** –

**let**  $?A = \text{insert } x \{s !! n \mid n. n \leq i\}$

**from** *assms* **obtain**  $j$  **where**  $*$ :  $\text{Sup\_fin } ?A < s !! j$

**by** (*auto simp: increasing\_def*)

**then have**  $x < s !! j$

**by** (*rule order.strict\_trans1*[rotated]) (*auto intro: Sup\_fin.coboundedI*)

**moreover have**  $i < j$

**proof** (*rule ccontr*)

**assume**  $\neg i < j$

**then have**  $s !! j \in ?A$  **by** (*auto simp: not\_less*)

**then have**  $s !! j \leq \text{Sup\_fin } ?A$

**by** (*auto intro: Sup\_fin.coboundedI*)

**with**  $*$  **show** *False* **by** *simp*

**qed**

**ultimately show** *?thesis* **by** *blast*

**qed**

**lemma** *sincreasing\_siterate\_nat*[*simp*]:

**fixes**  $n :: \text{nat}$

**assumes**  $(\bigwedge n. n < f n)$

**shows** *sincreasing* (*siterate*  $f n$ )

**unfolding** *increasing\_def* **proof**

**fix**  $x$

**show**  $\exists i. x < \text{siterate } f n !! i$

**proof** (*induction*  $x$ )

**case**  $0$

**have**  $0 < \text{siterate } f n !! 1$

**using** *order.strict\_trans1*[*OF le0 assms*] **by** *simp*

**then show** *?case* ..

**next**

**case** (*Suc*  $x$ )

**then obtain**  $i$  **where**  $x < \text{siterate } f n !! i$  ..

**then have** *Suc*  $x < \text{siterate } f n !! \text{Suc } i$

**using** *order.strict\_trans1*[*OF \_ assms*] **by** (*simp del: snth.simps*)

**then show** *?case* ..

**qed**

**qed**

**lemma** *sincreasing\_stl*: *sincreasing*  $s \implies \text{sincreasing } (\text{stl } s)$  **for**  $s :: 'a :: \text{semilattice\_sup}$  *stream*

**by** (*auto 0 4 simp: gr0\_conv\_Suc intro!: increasingI dest: increasing\_grD*[of  $s 0$ ])

**typedef**  $'a$  *trace* =  $\{s :: ('a \text{ set} \times \text{nat}) \text{ stream}. \text{sorted } (\text{smap } \text{snd } s) \wedge \text{increasing } (\text{smap } \text{snd } s)\}$

**by** (*intro exI*[of *\_ smap*] ( $\lambda i. (\{\}, i)$ ) *nats*)

(*auto simp: stream.map\_comp stream.map\_ident cong: stream.map\_cong*)

**setup\_lifting** *type\_definition\_trace*

**lift\_definition**  $\Gamma :: 'a \text{ trace} \Rightarrow \text{nat} \Rightarrow 'a \text{ set}$  **is**  
 $\lambda s i. \text{fst } (s !! i) .$

**lift\_definition**  $\tau :: 'a \text{ trace} \Rightarrow \text{nat} \Rightarrow \text{nat}$  **is**  
 $\lambda s i. \text{snd } (s !! i) .$

**lemma** *stream\_eq\_iff*:  $s = s' \iff (\forall n. s !! n = s' !! n)$   
**by** (*metis stream.map\_cong0 stream\_smap\_nats*)

**lemma** *trace\_eqI*:  $(\bigwedge i. \Gamma \sigma i = \Gamma \sigma' i) \implies (\bigwedge i. \tau \sigma i = \tau \sigma' i) \implies \sigma = \sigma'$   
**by** *transfer (auto simp: stream\_eq\_iff intro!: prod\_eqI)*

**lemma**  $\tau\_mono[simp]$ :  $i \leq j \implies \tau s i \leq \tau s j$   
**by** *transfer (auto simp: sorted\_iff\_mono)*

**lemma** *ex\_le\_τ*:  $\exists j \geq i. x \leq \tau s j$   
**by** (*transfer fixing: i x*) (*auto dest!: sincreasing\_grD[of \_ i x] less\_imp\_le*)

**lemma** *le\_τ\_less*:  $\tau \sigma i \leq \tau \sigma j \implies j < i \implies \tau \sigma i = \tau \sigma j$   
**by** (*simp add: antisym*)

**lemma** *less\_τD*:  $\tau \sigma i < \tau \sigma j \implies i < j$   
**by** (*meson τ\_mono less\_le\_not\_le not\_le\_imp\_less*)

**abbreviation**  $\Delta s i \equiv \tau s i - \tau s (i - 1)$

**lift\_definition** *map\_Γ* ::  $('a \text{ set} \Rightarrow 'b \text{ set}) \Rightarrow 'a \text{ trace} \Rightarrow 'b \text{ trace}$  **is**  
 $\lambda f s. \text{smap } (\lambda(x, i). (f x, i)) s$   
**by** (*auto simp: stream.map\_comp prod.case\_eq\_if cong: stream.map\_cong*)

**lemma**  $\Gamma\_map\_Γ[simp]$ :  $\Gamma (\text{map}_\Gamma f s) i = f (\Gamma s i)$   
**by** *transfer (simp add: prod.case\_eq\_if)*

**lemma**  $\tau\_map\_Γ[simp]$ :  $\tau (\text{map}_\Gamma f s) i = \tau s i$   
**by** *transfer (simp add: prod.case\_eq\_if)*

**lemma** *map\_Γ\_id[simp]*:  $\text{map}_\Gamma \text{id } s = s$   
**by** *transfer (simp add: stream.map\_id)*

**lemma** *map\_Γ\_comp*:  $\text{map}_\Gamma g (\text{map}_\Gamma f s) = \text{map}_\Gamma (g \circ f) s$   
**by** *transfer (simp add: stream.map\_comp comp\_def prod.case\_eq\_if case\_prod\_beta')*

**lemma** *map\_Γ\_cong*:  $\sigma_1 = \sigma_2 \implies (\bigwedge x. f_1 x = f_2 x) \implies \text{map}_\Gamma f_1 \sigma_1 = \text{map}_\Gamma f_2 \sigma_2$   
**by** *transfer (auto intro!: stream.map\_cong)*

## 1.2 Finite trace prefixes

**typedef**  $'a \text{ prefix} = \{p :: ('a \text{ set} \times \text{nat}) \text{ list. sorted } (\text{map } \text{snd } p)\}$   
**by** (*auto intro!: exI[of \_ []]*)

**setup\_lifting** *type\_definition\_prefix*

**lift\_definition** *pmap\_Γ* ::  $('a \text{ set} \Rightarrow 'b \text{ set}) \Rightarrow 'a \text{ prefix} \Rightarrow 'b \text{ prefix}$  **is**  
 $\lambda f. \text{map } (\lambda(x, i). (f x, i))$   
**by** (*simp add: split\_beta comp\_def*)

**lift\_definition** *last\_ts* ::  $'a \text{ prefix} \Rightarrow \text{nat}$  **is**

$\lambda p. (\text{case } p \text{ of } [] \Rightarrow 0 \mid \_ \Rightarrow \text{snd } (\text{last } p)) .$

**lift\_definition** *first\_ts* :: *nat*  $\Rightarrow$  '*a prefix*  $\Rightarrow$  *nat* **is**  
 $\lambda n p. (\text{case } p \text{ of } [] \Rightarrow n \mid \_ \Rightarrow \text{snd } (\text{hd } p)) .$

**lift\_definition** *pnil* :: '*a prefix* **is** [] **by** *simp*

**lift\_definition** *plen* :: '*a prefix*  $\Rightarrow$  *nat* **is** *length* .

**lift\_definition** *psnoc* :: '*a prefix*  $\Rightarrow$  '*a set*  $\times$  *nat*  $\Rightarrow$  '*a prefix* **is**  
 $\lambda p x. \text{if } (\text{case } p \text{ of } [] \Rightarrow 0 \mid \_ \Rightarrow \text{snd } (\text{last } p)) \leq \text{snd } x \text{ then } p @ [x] \text{ else } []$

**proof** (*goal\_cases sorted\_psnoc*)

**case** (*sorted\_psnoc* *p x*)

**then show** ?*case*

**by** (*induction* *p*) (*auto split: if\_splits list.splits*)

**qed**

**instantiation** *prefix* :: (*type*) **order begin**

**lift\_definition** *less\_eq\_prefix* :: '*a prefix*  $\Rightarrow$  '*a prefix*  $\Rightarrow$  *bool* **is**  
 $\lambda p q. \exists r. q = p @ r .$

**definition** *less\_prefix* :: '*a prefix*  $\Rightarrow$  '*a prefix*  $\Rightarrow$  *bool* **where**  
*less\_prefix* *x y* = ( $x \leq y \wedge \neg y \leq x$ )

**instance**

**proof** (*standard, goal\_cases less\_refl\_trans\_antisym*)

**case** (*less* *x y*)

**then show** ?*case* **unfolding** *less\_prefix\_def* ..

**next**

**case** (*refl* *x*)

**then show** ?*case* **by** *transfer auto*

**next**

**case** (*trans* *x y z*)

**then show** ?*case* **by** *transfer auto*

**next**

**case** (*antisym* *x y*)

**then show** ?*case* **by** *transfer auto*

**qed**

**end**

**lemma** *psnoc\_inject*[*simp*]:

$\text{last\_ts } p \leq \text{snd } x \Longrightarrow \text{last\_ts } q \leq \text{snd } y \Longrightarrow \text{psnoc } p \ x = \text{psnoc } q \ y \longleftrightarrow (p = q \wedge x = y)$

**by** *transfer auto*

**lift\_definition** *prefix\_of* :: '*a prefix*  $\Rightarrow$  '*a trace*  $\Rightarrow$  *bool* **is**  $\lambda p s. \text{stake } (\text{length } p) \ s = p .$

**lemma** *prefix\_of\_pnil*[*simp*]: *prefix\_of* *pnil*  $\sigma$

**by** *transfer auto*

**lemma** *plen\_pnil*[*simp*]: *plen* *pnil* = 0

**by** *transfer auto*

**lemma** *prefix\_of\_pmap\_Γ*[*simp*]: *prefix\_of*  $\pi \ \sigma \Longrightarrow \text{prefix\_of } (\text{pmap}_\Gamma \ f \ \pi) \ (\text{map}_\Gamma \ f \ \sigma)$

**by** *transfer auto*

**lemma** *plen\_mono*:  $\pi \leq \pi' \Longrightarrow \text{plen } \pi \leq \text{plen } \pi'$

by transfer auto

**lemma** prefix\_of\_psnocE: prefix\_of (psnoc p x) s  $\implies$  last\_ts p  $\leq$  snd x  $\implies$   
(prefix\_of p s  $\implies$   $\Gamma$  s (plen p) = fst x  $\implies$   $\tau$  s (plen p) = snd x  $\implies$  P)  $\implies$  P  
by transfer (simp del: stake.simps add: stake\_Suc)

**lemma** le\_pnil[simp]: pnul  $\leq$   $\pi$   
by transfer auto

**lift\_definition** take\_prefix :: nat  $\Rightarrow$  'a trace  $\Rightarrow$  'a prefix is stake  
by (auto dest: sorted\_stake)

**lemma** plen\_take\_prefix[simp]: plen (take\_prefix i  $\sigma$ ) = i  
by transfer auto

**lemma** plen\_psnoc[simp]: last\_ts  $\pi \leq$  snd x  $\implies$  plen (psnoc  $\pi$  x) = plen  $\pi$  + 1  
by transfer auto

**lemma** prefix\_of\_take\_prefix[simp]: prefix\_of (take\_prefix i  $\sigma$ )  $\sigma$   
by transfer auto

**lift\_definition** pdrop :: nat  $\Rightarrow$  'a prefix  $\Rightarrow$  'a prefix is drop  
by (auto simp: drop\_map[symmetric] sorted\_wrt\_drop)

**lemma** pdrop\_0[simp]: pdrop 0  $\pi$  =  $\pi$   
by transfer auto

**lemma** prefix\_of\_antimono:  $\pi \leq \pi' \implies$  prefix\_of  $\pi' s \implies$  prefix\_of  $\pi s$   
by transfer (auto simp del: stake\_add simp add: stake\_add[symmetric])

**lemma** prefix\_of\_imp\_linear: prefix\_of  $\pi \sigma \implies$  prefix\_of  $\pi' \sigma \implies \pi \leq \pi' \vee \pi' \leq \pi$   
**proof** transfer

fix  $\pi \pi'$  and  $\sigma$  :: ('a set  $\times$  nat) stream

assume assms: stake (length  $\pi$ )  $\sigma$  =  $\pi$  stake (length  $\pi'$ )  $\sigma$  =  $\pi'$

show  $(\exists r. \pi' = \pi @ r) \vee (\exists r. \pi = \pi' @ r)$

**proof** (cases length  $\pi$  length  $\pi'$  rule: le\_cases)

case le

then have  $\pi' = \text{take } (\text{length } \pi) \pi' @ \text{drop } (\text{length } \pi) \pi'$

by simp

moreover have  $\text{take } (\text{length } \pi) \pi' = \pi$

using assms le by (metis min.absorb1 take\_stake)

ultimately show ?thesis by auto

next

case ge

then have  $\pi = \text{take } (\text{length } \pi') \pi @ \text{drop } (\text{length } \pi') \pi$

by simp

moreover have  $\text{take } (\text{length } \pi') \pi = \pi'$

using assms ge by (metis min.absorb1 take\_stake)

ultimately show ?thesis by auto

qed

qed

**lemma** ex\_prefix\_of:  $\exists s. \text{prefix\_of } p s$

**proof** (transfer, intro bexI CollectI conjI)

fix p :: ('a set  $\times$  nat) list

assume \*: sorted (map snd p)

let  $?\sigma = p @- \text{smap } (\text{Pair undefined}) (\text{fromN } (\text{snd } (\text{last } p)))$

show stake (length p)  $?\sigma = p$  by (simp add: stake\_shift)

```

have le_last: snd (p ! i) ≤ snd (last p) if i < length p for i
  using sorted_nth_mono[OF *, of i length p - 1] that
  by (cases p) (auto simp: last_conv_nth nth_Cons')
with * show ssorted (smap snd ?σ)
  by (force simp: ssorted_iff_mono sorted_iff_nth_mono shift_snth)
show sincreasing (smap snd ?σ)
proof (rule sincreasingI)
  fix x
  have x < smap snd ?σ !! Suc (length p + x)
  by simp (metis Suc_pred add.commute diff_Suc_Suc length_greater_0_conv less_add_Suc1 less_diff_conv)
  then show ∃ i. x < smap snd ?σ !! i ..
qed
qed

lemma τ_prefix_conv: prefix_of p s ⇒ prefix_of p s' ⇒ i < plen p ⇒ τ s i = τ s' i
  by transfer (simp add: stake_nth[symmetric])

lemma Γ_prefix_conv: prefix_of p s ⇒ prefix_of p s' ⇒ i < plen p ⇒ Γ s i = Γ s' i
  by transfer (simp add: stake_nth[symmetric])

lemma sincreasing_sdrop:
  fixes s :: ('a :: semilattice_sup) stream
  assumes sincreasing s
  shows sincreasing (sdrop n s)
proof (rule sincreasingI)
  fix x
  obtain i where n < i and x < s !! i
  using sincreasing_grD[OF assms] by blast
  then have x < sdrop n s !! (i - n)
  by (simp add: sdrop_snth)
  then show ∃ i. x < sdrop n s !! i ..
qed

lemma sorted_shift:
  sorted (xs @- s) = (sorted xs ∧ ssorted s ∧ (∀ x∈set xs. ∀ y∈sset s. x ≤ y))
proof safe
  assume *: ssorted (xs @- s)
  then show sorted xs
  by (auto simp: ssorted_iff_mono shift_snth sorted_iff_nth_mono split: if_splits)
  from ssorted_sdrop[OF *, of length xs] show ssorted s
  by (auto simp: sdrop_shift)
  fix x y assume x ∈ set xs y ∈ sset s
  then obtain i j where i < length xs xs ! i = x s !! j = y
  by (auto simp: set_conv_nth sset_range)
  with ssorted_monoD[OF *, of i j + length xs] show x ≤ y by auto
next
  assume sorted xs ssorted s ∀ x∈set xs. ∀ y∈sset s. x ≤ y
  then show sorted (xs @- s)
  proof (coinduction arbitrary: xs s)
  case (ssorted xs s)
  with ⟨ssorted s⟩ show ?case
  by (subst (asm) ssorted.simps) (auto 0 4 simp: neq_Nil_conv shd_sset intro: exI[of _ _ # _])
qed
qed

lemma sincreasing_shift:
  assumes sincreasing s
  shows sincreasing (xs @- s)

```

**proof** (rule *sincreasingI*)

**fix**  $x$   
**from** *assms* **obtain**  $i$  **where**  $x < s !! i$   
**unfolding** *sincreasing\_def* **by** *blast*  
**then have**  $x < (xs @- s) !! (\text{length } xs + i)$   
**by** *simp*  
**then show**  $\exists i. x < (xs @- s) !! i ..$

**qed**

**lift\_definition** *replace\_prefix* :: 'a prefix  $\Rightarrow$  'a trace  $\Rightarrow$  'a trace **is**

$\lambda \pi \sigma. \text{if } \text{ssorted } (\text{smap } \text{snd } (\pi @- \text{sdrop } (\text{length } \pi) \sigma)) \text{ then}$   
 $\pi @- \text{sdrop } (\text{length } \pi) \sigma \text{ else } \text{smap } (\lambda i. (\{\}, i)) \text{ nats}$

**by** (*auto split: if\_splits simp: stream.map\_comp stream.map\_ident sdrop\_smap[symmetric]*  
*simp del: sdrop\_smap intro!: increasing\_shift increasing\_sdrop cong: stream.map\_comp*)

**lemma** *prefix\_of\_replace\_prefix*:

*prefix\_of* (*pmap* $_{\Gamma}$   $f$   $\pi$ )  $\sigma \Longrightarrow \text{prefix\_of } \pi (\text{replace\_prefix } \pi \sigma)$

**proof** (*transfer; safe; goal\_cases*)

**case** ( $1 f \pi \sigma$ )

**then show** ?*case*

**by** (*subst (asm) (2) stake\_sdrop[symmetric, of \_ length  $\pi$ ]*)

(*auto 0 3 simp: sorted\_shift split\_beta o\_def stake\_shift sdrop\_smap[symmetric]*  
*sorted\_sdrop not\_le simp del: sdrop\_smap*)

**qed**

**lemma** *map\_ $_{\Gamma}$ \_replace\_prefix*:

$\forall x. f (f x) = f x \Longrightarrow \text{prefix\_of } (\text{pmap\_}\Gamma f \pi) \sigma \Longrightarrow \text{map\_}\Gamma f (\text{replace\_prefix } \pi \sigma) = \text{map\_}\Gamma f \sigma$

**proof** (*transfer; safe; goal\_cases*)

**case** ( $1 f \pi \sigma$ )

**then show** ?*case*

**by** (*subst (asm) (2) stake\_sdrop[symmetric, of  $\sigma$  length  $\pi$ ]*,

*subst (3) stake\_sdrop[symmetric, of  $\sigma$  length  $\pi$ ]*)

(*auto simp: sorted\_shift split\_beta o\_def stake\_shift sdrop\_smap[symmetric] sorted\_sdrop*  
*not\_le simp del: sdrop\_smap cong: map\_comp*)

**qed**

**lemma** *prefix\_of\_pmap\_ $_{\Gamma}$ \_D*:

**assumes** *prefix\_of* (*pmap* $_{\Gamma}$   $f$   $\pi$ )  $\sigma$

**shows**  $\exists \sigma'. \text{prefix\_of } \pi \sigma' \wedge \text{prefix\_of } (\text{pmap\_}\Gamma f \pi) (\text{map\_}\Gamma f \sigma')$

**proof** –

**from** *assms*(1) **obtain**  $\sigma'$  **where**  $1: \text{prefix\_of } \pi \sigma'$

**using** *ex\_prefix\_of* **by** *blast*

**then have** *prefix\_of* (*pmap* $_{\Gamma}$   $f$   $\pi$ ) (*map* $_{\Gamma}$   $f$   $\sigma'$ )

**by** *transfer simp*

**with**  $1$  **show** ?*thesis* **by** *blast*

**qed**

**lemma** *prefix\_of\_map\_ $_{\Gamma}$ \_D*:

**assumes** *prefix\_of*  $\pi' (\text{map\_}\Gamma f \sigma)$

**shows**  $\exists \pi''. \pi' = \text{pmap\_}\Gamma f \pi'' \wedge \text{prefix\_of } \pi'' \sigma$

**using** *assms*

**by** *transfer (auto intro!: exI[of \_ stake (length \_) \_] elim: sym dest: sorted\_stake)*

**lift\_definition** *pts* :: 'a prefix  $\Rightarrow$  nat list **is** *map snd* .

**lemma** *pts\_pmap\_ $_{\Gamma}$ [simp]*: *pts* (*pmap* $_{\Gamma}$   $f$   $\pi$ ) = *pts*  $\pi$

**by** (*transfer fixing: f (simp add: split\_beta)*)

## 2 Finite tables

**primrec** *tabulate* :: (nat  $\Rightarrow$  'a)  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  'a list **where**  
*tabulate* f x 0 = []  
| *tabulate* f x (Suc n) = f x # *tabulate* f (Suc x) n

**lemma** *tabulate\_alt*: *tabulate* f x n = map f [x ..< x + n]  
**by** (induct n arbitrary: x) (auto simp: not\_le Suc\_le\_eq upt\_rec)

**lemma** *length\_tabulate[simp]*: length (*tabulate* f x n) = n  
**by** (induction n arbitrary: x) simp\_all

**lemma** *map\_tabulate[simp]*: map f (*tabulate* g x n) = *tabulate* ( $\lambda x. f (g x)$ ) x n  
**by** (induction n arbitrary: x) simp\_all

**lemma** *nth\_tabulate[simp]*: k < n  $\implies$  *tabulate* f x n ! k = f (x + k)  
**proof** (induction n arbitrary: x k)  
**case** (Suc n)  
**then show** ?case **by** (cases k) simp\_all  
**qed** simp

**type\_synonym** 'a tuple = 'a option list  
**type\_synonym** 'a table = 'a tuple set

**definition** *wf\_tuple* :: nat  $\Rightarrow$  nat set  $\Rightarrow$  'a tuple  $\Rightarrow$  bool **where**  
*wf\_tuple* n V x  $\longleftrightarrow$  length x = n  $\wedge$  ( $\forall i < n. x ! i = \text{None} \longleftrightarrow i \notin V$ )

**definition** *table* :: nat  $\Rightarrow$  nat set  $\Rightarrow$  'a table  $\Rightarrow$  bool **where**  
*table* n V X  $\longleftrightarrow$  ( $\forall x \in X. \text{wf\_tuple } n \ V \ x$ )

**definition** *empty\_table* = {}

**definition** *unit\_table* n = {replicate n None}

**definition** *singleton\_table* n i x = {*tabulate* ( $\lambda j. \text{if } i = j \text{ then } \text{Some } x \text{ else } \text{None}$ ) 0 n}

**lemma** *in\_empty\_table[simp]*:  $\neg x \in \text{empty\_table}$   
**unfolding** *empty\_table\_def* **by** simp

**lemma** *empty\_table[simp]*: *table* n V *empty\_table*  
**unfolding** *table\_def* *empty\_table\_def* **by** simp

**lemma** *unit\_table\_wf\_tuple[simp]*: V = {}  $\implies$  x  $\in$  *unit\_table* n  $\implies$  *wf\_tuple* n V x  
**unfolding** *unit\_table\_def* *wf\_tuple\_def* **by** simp

**lemma** *unit\_table[simp]*: V = {}  $\implies$  *table* n V (*unit\_table* n)  
**unfolding** *table\_def* **by** simp

**lemma** *in\_unit\_table*: v  $\in$  *unit\_table* n  $\longleftrightarrow$  *wf\_tuple* n {} v  
**unfolding** *unit\_table\_def* *wf\_tuple\_def* **by** (auto intro!: nth\_equalityI)

**lemma** *singleton\_table\_wf\_tuple[simp]*: V = {i}  $\implies$  x  $\in$  *singleton\_table* n i z  $\implies$  *wf\_tuple* n V x  
**unfolding** *singleton\_table\_def* *wf\_tuple\_def* **by** simp

**lemma** *singleton\_table[simp]*: V = {i}  $\implies$  *table* n V (*singleton\_table* n i z)  
**unfolding** *table\_def* **by** simp

**lemma** *table\_Un[simp]*: *table* n V X  $\implies$  *table* n V Y  $\implies$  *table* n V (X  $\cup$  Y)  
**unfolding** *table\_def* **by** auto

**lemma** *wf\_tuple\_length*:  $wf\_tuple\ n\ V\ x \implies length\ x = n$   
**unfolding** *wf\_tuple\_def* **by** *simp*

**fun** *join1* :: 'a tuple × 'a tuple ⇒ 'a tuple option **where**  
*join1* ([], []) = Some []  
| *join1* (None # xs, None # ys) = map\_option (Cons None) (*join1* (xs, ys))  
| *join1* (Some x # xs, None # ys) = map\_option (Cons (Some x)) (*join1* (xs, ys))  
| *join1* (None # xs, Some y # ys) = map\_option (Cons (Some y)) (*join1* (xs, ys))  
| *join1* (Some x # xs, Some y # ys) = (if x = y  
then map\_option (Cons (Some x)) (*join1* (xs, ys))  
else None)  
| *join1* \_ = None

**definition** *join* :: 'a table ⇒ bool ⇒ 'a table ⇒ 'a table **where**  
*join* A pos B = (if pos then Option.these (*join1* ' (A × B))  
else A - Option.these (*join1* ' (A × B)))

**lemma** *join\_True\_code*[code]:  $join\ A\ True\ B = (\bigcup a \in A. \bigcup b \in B. set\_option\ (join1\ (a, b)))$   
**unfolding** *join\_def* **by** (force *simp*: Option.these\_def image\_iff)

**lemma** *join\_False\_alt*:  $join\ X\ False\ Y = X - join\ X\ True\ Y$   
**unfolding** *join\_def* **by** *auto*

**lemma** *self\_join1*:  $join1\ (xs, ys) \neq Some\ xs \implies join1\ (zs, ys) \neq Some\ xs$   
**by** (induct (zs, ys) arbitrary: zs ys xs rule: *join1.induct*; auto; auto)

**lemma** *join\_False\_code*[code]:  $join\ A\ False\ B = \{a \in A. \forall b \in B. join1\ (a, b) \neq Some\ a\}$   
**unfolding** *join\_False\_alt* *join\_True\_code*  
**by** (auto *simp*: Option.these\_def image\_iff dest: *self\_join1*)

**lemma** *wf\_tuple\_Nil*[simp]:  $wf\_tuple\ n\ A\ [] = (n = 0)$   
**unfolding** *wf\_tuple\_def* **by** *auto*

**lemma** *Suc\_pred'*:  $Suc\ (x - Suc\ 0) = (case\ x\ of\ 0 \Rightarrow Suc\ 0\ |\ _ \Rightarrow x)$   
**by** (auto *split*: nat.splits)

**lemma** *wf\_tuple\_Cons*[simp]:  
 $wf\_tuple\ n\ A\ (x \# xs) \longleftrightarrow ((if\ x = None\ then\ 0 \notin A\ else\ 0 \in A) \wedge$   
 $(\exists m. n = Suc\ m \wedge wf\_tuple\ m\ ((\lambda x. x - 1) ' (A - \{0\}))\ xs))$   
**unfolding** *wf\_tuple\_def*  
**by** (auto 0 3 *simp*: nth\_Cons image\_iff Ball\_def gr0\_conv\_Suc Suc\_pred' *split*: nat.splits)

**lemma** *join1\_wf\_tuple*:  
 $join1\ (v1, v2) = Some\ v \implies wf\_tuple\ n\ A\ v1 \implies wf\_tuple\ n\ B\ v2 \implies wf\_tuple\ n\ (A \cup B)\ v$   
**by** (induct (v1, v2) arbitrary: n v v1 v2 A B rule: *join1.induct*)  
(auto *simp*: image\_Un Un\_Diff *split*: if\_splits)

**lemma** *join\_wf\_tuple*:  $x \in join\ X\ b\ Y \implies$   
 $\forall v \in X. wf\_tuple\ n\ A\ v \implies \forall v \in Y. wf\_tuple\ n\ B\ v \implies (\neg b \implies B \subseteq A) \implies A \cup B = C \implies$   
 $wf\_tuple\ n\ C\ x$   
**unfolding** *join\_def*  
**by** (fastforce *simp*: Option.these\_def image\_iff sup\_absorb1 dest: *join1\_wf\_tuple* *split*: if\_splits)

**lemma** *join\_table*:  $table\ n\ A\ X \implies table\ n\ B\ Y \implies (\neg b \implies B \subseteq A) \implies A \cup B = C \implies$   
 $table\ n\ C\ (join\ X\ b\ Y)$   
**unfolding** *table\_def* **by** (auto *elim!*: *join\_wf\_tuple*)

**lemma** *wf\_tuple\_Suc*:  $wf\_tuple (Suc\ m)\ A\ a \longleftrightarrow a \neq [] \wedge$   
 $wf\_tuple\ m\ ((\lambda x. x - 1)\ ' (A - \{0\}))\ (tl\ a) \wedge (0 \in A \longleftrightarrow hd\ a \neq None)$   
**by** (*cases a*) (*auto simp: nth\_Cons image\_iff split: nat.splits*)

**lemma** *table\_project*:  $table (Suc\ n)\ A\ X \implies table\ n\ ((\lambda x. x - Suc\ 0)\ ' (A - \{0\}))\ (tl\ ' X)$   
**unfolding** *table\_def*  
**by** (*auto simp: wf\_tuple\_Suc*)

**definition** *restrict where*

*restrict A v = map (\lambda i. if i \in A then v ! i else None) [0 ..< length v]*

**lemma** *restrict\_Nil[simp]*:  $restrict\ A\ [] = []$   
**unfolding** *restrict\_def* **by** *auto*

**lemma** *restrict\_Cons[simp]*:  $restrict\ A\ (x \# xs) =$   
*(if 0 \in A then x \# restrict ((\lambda x. x - 1)\ ' (A - \{0\})) xs else None \# restrict ((\lambda x. x - 1)\ ' A) xs)*  
**unfolding** *restrict\_def*  
**by** (*auto simp: map\_upt\_Suc image\_iff Suc\_pred' Ball\_def simp del: upt\_Suc split: nat.splits*)

**lemma** *wf\_tuple\_restrict*:  $wf\_tuple\ n\ B\ v \implies A \cap B = C \implies wf\_tuple\ n\ C\ (restrict\ A\ v)$   
**unfolding** *restrict\_def wf\_tuple\_def* **by** *auto*

**lemma** *wf\_tuple\_restrict\_simple*:  $wf\_tuple\ n\ B\ v \implies A \subseteq B \implies wf\_tuple\ n\ A\ (restrict\ A\ v)$   
**unfolding** *restrict\_def wf\_tuple\_def* **by** *auto*

**lemma** *nth\_restrict*:  $i \in A \implies i < length\ v \implies restrict\ A\ v ! i = v ! i$   
**unfolding** *restrict\_def* **by** *auto*

**lemma** *restrict\_eq\_Nil[simp]*:  $restrict\ A\ v = [] \longleftrightarrow v = []$   
**unfolding** *restrict\_def* **by** *auto*

**lemma** *length\_restrict[simp]*:  $length\ (restrict\ A\ v) = length\ v$   
**unfolding** *restrict\_def* **by** *auto*

**lemma** *join1\_Some\_restrict*:

**fixes**  $x\ y :: 'a\ tuple$

**assumes**  $wf\_tuple\ n\ A\ x\ wf\_tuple\ n\ B\ y$

**shows**  $join1\ (x, y) = Some\ z \longleftrightarrow wf\_tuple\ n\ (A \cup B)\ z \wedge restrict\ A\ z = x \wedge restrict\ B\ z = y$

**using** *assms*

**proof** (*induct (x, y) arbitrary: n x y z A B rule: join1.induct*)

**case** (*2 xs ys*)

**then show** *?case*

**by** (*cases z*) (*auto 4 0 simp: image\_Un Un\_Diff*)**+**

**next**

**case** (*3 x xs ys*)

**then show** *?case*

**by** (*cases z*) (*auto 4 0 simp: image\_Un Un\_Diff*)**+**

**next**

**case** (*4 xs y ys*)

**then show** *?case*

**by** (*cases z*) (*auto 4 0 simp: image\_Un Un\_Diff*)**+**

**next**

**case** (*5 x xs y ys*)

**then show** *?case*

**by** (*cases z*) (*auto 4 0 simp: image\_Un Un\_Diff*)**+**

**qed** *auto*

**lemma restrict\_idle:**  $wf\_tuple\ n\ A\ v \implies restrict\ A\ v = v$   
**by** (induct v arbitrary: n A) (auto split: if\_splits)

**lemma map\_the\_restrict:**

$i \in A \implies map\ the\ (restrict\ A\ v)\ !\ i = map\ the\ v\ !\ i$   
**by** (induct v arbitrary: A i) (auto simp: nth\_Cons' gr0\_conv\_Suc split: option.splits)

**lemma join\_restrict:**

**fixes** X Y :: 'a tuple set  
**assumes**  $\bigwedge v. v \in X \implies wf\_tuple\ n\ A\ v \wedge \bigwedge v. v \in Y \implies wf\_tuple\ n\ B\ v \neg b \implies B \subseteq A$   
**shows**  $v \in join\ X\ b\ Y \longleftrightarrow$   
 $wf\_tuple\ n\ (A \cup B)\ v \wedge restrict\ A\ v \in X \wedge (if\ b\ then\ restrict\ B\ v \in Y\ else\ restrict\ B\ v \notin Y)$   
**by** (auto 4 4 simp: join\_def Option.these\_def image\_iff assms wf\_tuple\_restrict sup\_absorb1 restrict\_idle  
restrict\_idle[OF assms(1)] elim: assms  
dest: join1\_Some\_restrict[OF assms(1,2), THEN iffD1, rotated -1]  
dest!: spec[of\_ Some v]  
intro!: exI[of\_ Some v] join1\_Some\_restrict[THEN iffD2, symmetric] bexI[rotated])

**lemma join\_restrict\_table:**

**assumes** table n A X table n B Y  $\neg b \implies B \subseteq A$   
**shows**  $v \in join\ X\ b\ Y \longleftrightarrow$   
 $wf\_tuple\ n\ (A \cup B)\ v \wedge restrict\ A\ v \in X \wedge (if\ b\ then\ restrict\ B\ v \in Y\ else\ restrict\ B\ v \notin Y)$   
**using** assms **unfolding** table\_def  
**by** (simp add: join\_restrict)

**lemma join\_restrict\_annotated:**

**fixes** X Y :: 'a tuple set  
**assumes**  $\neg b = simp \implies B \subseteq A$   
**shows**  $join\ \{v. wf\_tuple\ n\ A\ v \wedge P\ v\}\ b\ \{v. wf\_tuple\ n\ B\ v \wedge Q\ v\} =$   
 $\{v. wf\_tuple\ n\ (A \cup B)\ v \wedge P\ (restrict\ A\ v) \wedge (if\ b\ then\ Q\ (restrict\ B\ v)\ else\ \neg\ Q\ (restrict\ B\ v))\}$   
**using** assms  
**by** (intro set\_eqI, subst join\_restrict) (auto simp: wf\_tuple\_restrict\_simple simp\_implies\_def)

**lemma in\_joinI:** table n A X  $\implies$  table n B Y  $\implies$  ( $\neg b \implies B \subseteq A$ )  $\implies$   $wf\_tuple\ n\ (A \cup B)\ v \implies$   
 $restrict\ A\ v \in X \implies (b \implies restrict\ B\ v \in Y) \implies (\neg b \implies restrict\ B\ v \notin Y) \implies v \in join\ X\ b\ Y$   
**unfolding** table\_def  
**by** (subst join\_restrict) (auto)

**lemma in\_joinE:**  $v \in join\ X\ b\ Y \implies$  table n A X  $\implies$  table n B Y  $\implies$  ( $\neg b \implies B \subseteq A$ )  $\implies$   
 $(wf\_tuple\ n\ (A \cup B)\ v \implies restrict\ A\ v \in X \implies if\ b\ then\ restrict\ B\ v \in Y\ else\ restrict\ B\ v \notin Y \implies$   
 $P) \implies P$   
**unfolding** table\_def  
**by** (subst (asm) join\_restrict) (auto)

**definition** qtable :: nat  $\Rightarrow$  nat set  $\Rightarrow$  ('a tuple  $\Rightarrow$  bool)  $\Rightarrow$  ('a tuple  $\Rightarrow$  bool)  $\Rightarrow$   
'a table  $\Rightarrow$  bool **where**

qtable n A P Q X  $\longleftrightarrow$  table n A X  $\wedge$  ( $\forall x. (x \in X \wedge P\ x \longrightarrow Q\ x) \wedge (wf\_tuple\ n\ A\ x \wedge P\ x \wedge Q\ x$   
 $\longrightarrow x \in X)$ )

**abbreviation** wf\_table **where**

$wf\_table\ n\ A\ Q\ X \equiv qtable\ n\ A\ (\lambda_.\ True)\ Q\ X$

**lemma wf\_table\_iff:**  $wf\_table\ n\ A\ Q\ X \longleftrightarrow (\forall x. x \in X \longleftrightarrow (Q\ x \wedge wf\_tuple\ n\ A\ x))$   
**unfolding** qtable\_def table\_def **by** auto

**lemma table\_wf\_table:** table n A X =  $wf\_table\ n\ A\ (\lambda v. v \in X)\ X$   
**unfolding** table\_def wf\_table\_iff **by** auto

**lemma** *qtableI*:  $table\ n\ A\ X \implies$   
 $(\bigwedge x. x \in X \implies wf\_tuple\ n\ A\ x \implies P\ x \implies Q\ x) \implies$   
 $(\bigwedge x. wf\_tuple\ n\ A\ x \implies P\ x \implies Q\ x \implies x \in X) \implies$   
 $qtable\ n\ A\ P\ Q\ X$   
**unfolding** *qtable\_def table\_def* **by** *auto*

**lemma** *in\_qtableI*:  $qtable\ n\ A\ P\ Q\ X \implies wf\_tuple\ n\ A\ x \implies P\ x \implies Q\ x \implies x \in X$   
**unfolding** *qtable\_def* **by** *blast*

**lemma** *in\_qtableE*:  $qtable\ n\ A\ P\ Q\ X \implies x \in X \implies P\ x \implies (wf\_tuple\ n\ A\ x \implies Q\ x \implies R) \implies R$   
**unfolding** *qtable\_def table\_def* **by** *blast*

**lemma** *qtable\_empty*:  $(\bigwedge x. wf\_tuple\ n\ A\ x \implies P\ x \implies Q\ x \implies False) \implies qtable\ n\ A\ P\ Q\ empty\_table$   
**unfolding** *qtable\_def table\_def empty\_table\_def* **by** *auto*

**lemma** *qtable\_empty\_iff*:  $qtable\ n\ A\ P\ Q\ empty\_table = (\forall x. wf\_tuple\ n\ A\ x \longrightarrow P\ x \longrightarrow Q\ x \longrightarrow False)$   
**unfolding** *qtable\_def table\_def empty\_table\_def* **by** *auto*

**lemma** *qtable\_unit\_table*:  $(\bigwedge x. wf\_tuple\ n\ \{\} x \implies P\ x \implies Q\ x) \implies qtable\ n\ \{\} P\ Q\ (unit\_table\ n)$   
**unfolding** *qtable\_def table\_def in\_unit\_table* **by** *auto*

**lemma** *qtable\_union*:  $qtable\ n\ A\ P\ Q1\ X \implies qtable\ n\ A\ P\ Q2\ Y \implies$   
 $(\bigwedge x. wf\_tuple\ n\ A\ x \implies P\ x \implies Q\ x \longleftrightarrow Q1\ x \vee Q2\ x) \implies qtable\ n\ A\ P\ Q\ (X \cup Y)$   
**unfolding** *qtable\_def table\_def* **by** *blast*

**lemma** *qtable\_Union*:  $finite\ I \implies (\bigwedge i. i \in I \implies qtable\ n\ A\ P\ (Qi\ i)\ (Xi\ i)) \implies$   
 $(\bigwedge x. wf\_tuple\ n\ A\ x \implies P\ x \implies Q\ x \longleftrightarrow (\exists i \in I. Qi\ i\ x)) \implies qtable\ n\ A\ P\ Q\ (\bigcup i \in I. Xi\ i)$   
**proof** (*induct I arbitrary: Q rule: finite\_induct*)  
**case** (*insert i F*)  
**then show** *?case*  
**by** (*auto intro!: qtable\_union[where ?Q1.0 = Qi i and ?Q2.0 =  $\lambda x. \exists i \in F. Qi\ i\ x$ ]*)  
**qed** (*auto intro!: qtable\_empty[unfolded empty\_table\_def]*)

**lemma** *qtable\_join*:  
**assumes**  $qtable\ n\ A\ P\ Q1\ X\ qtable\ n\ B\ P\ Q2\ Y\ \neg b \implies B \subseteq A\ C = A \cup B$   
 $\bigwedge x. wf\_tuple\ n\ C\ x \implies P\ x \implies P\ (restrict\ A\ x) \wedge P\ (restrict\ B\ x)$   
 $\bigwedge x. b \implies wf\_tuple\ n\ C\ x \implies P\ x \implies Q\ x \longleftrightarrow Q1\ (restrict\ A\ x) \wedge Q2\ (restrict\ B\ x)$   
 $\bigwedge x. \neg b \implies wf\_tuple\ n\ C\ x \implies P\ x \implies Q\ x \longleftrightarrow Q1\ (restrict\ A\ x) \wedge \neg Q2\ (restrict\ B\ x)$   
**shows**  $qtable\ n\ C\ P\ Q\ (join\ X\ b\ Y)$   
**proof** (*rule qtableI*)  
**from** *assms(1-4)* **show**  $table\ n\ C\ (join\ X\ b\ Y)$   
**unfolding** *qtable\_def* **by** (*auto simp: join\_table*)  
**next**  
**fix** *x* **assume**  $x \in join\ X\ b\ Y\ wf\_tuple\ n\ C\ x\ P\ x$   
**with** *assms(1-3)* *assms(5-7)[of x]* **show**  $Q\ x$  **unfolding** *qtable\_def*  
**by** (*auto 0 2 simp: wf\_tuple\_restrict\_simple elim!: in\_joinE split: if\_splits*)  
**next**  
**fix** *x* **assume**  $wf\_tuple\ n\ C\ x\ P\ x\ Q\ x$   
**with** *assms(1-4)* *assms(5-7)[of x]* **show**  $x \in join\ X\ b\ Y$  **unfolding** *qtable\_def*  
**by** (*auto dest: wf\_tuple\_restrict\_simple intro!: in\_joinI[of n A X B Y]*)  
**qed**

**lemma** *qtable\_join\_fixed*:  
**assumes**  $qtable\ n\ A\ P\ Q1\ X\ qtable\ n\ B\ P\ Q2\ Y\ \neg b \implies B \subseteq A\ C = A \cup B$   
 $\bigwedge x. wf\_tuple\ n\ C\ x \implies P\ x \implies P\ (restrict\ A\ x) \wedge P\ (restrict\ B\ x)$   
**shows**  $qtable\ n\ C\ P\ (\lambda x. Q1\ (restrict\ A\ x) \wedge (if\ b\ then\ Q2\ (restrict\ B\ x)\ else\ \neg Q2\ (restrict\ B\ x)))$

(join X b Y)

by (rule qtable\_join[OF assms]) auto

**lemma** wf\_tuple\_cong:

assumes wf\_tuple n A v wf\_tuple n A w  $\forall x \in A. \text{map the } v ! x = \text{map the } w ! x$   
shows v = w

**proof** –

from assms(1,2) have length v = length w **unfolding** wf\_tuple\_def **by** simp  
from this assms **show** v = w

**proof** (induct v w arbitrary: n A rule: list\_induct2)

case (Cons x xs y ys)

let ?n = n - 1 **and** ?A = ( $\lambda x. x - 1$ ) ‘ (A - {0})

have \*: map the xs ! z = map the ys ! z **if** z  $\in$  ?A **for** z

using that Cons(5)[THEN bspec, of Suc z]

**by** (cases z) (auto simp: le\_Suc\_eq split: if\_splits)

**from** Cons(1,3–5) **show** ?case

**by** (auto intro!: Cons(2)[of ?n ?A] \* split: if\_splits)

**qed** simp

**qed**

**definition** mem\_restr :: 'a list set  $\Rightarrow$  'a tuple  $\Rightarrow$  bool **where**

mem\_restr A x  $\longleftrightarrow$  ( $\exists y \in A. \text{list\_all2 } (\lambda a b. a \neq \text{None} \longrightarrow a = \text{Some } b) x y$ )

**lemma** mem\_restrI:  $y \in A \Longrightarrow \text{length } y = n \Longrightarrow \text{wf\_tuple } n V x \Longrightarrow \forall i \in V. x ! i = \text{Some } (y ! i) \Longrightarrow$   
mem\_restr A x

**unfolding** mem\_restr\_def wf\_tuple\_def **by** (force simp add: list\_all2\_conv\_all\_nth)

**lemma** mem\_restrE: mem\_restr A x  $\Longrightarrow \text{wf\_tuple } n V x \Longrightarrow \forall i \in V. i < n \Longrightarrow$

( $\bigwedge y. y \in A \Longrightarrow \forall i \in V. x ! i = \text{Some } (y ! i) \Longrightarrow P$ )  $\Longrightarrow P$

**unfolding** mem\_restr\_def wf\_tuple\_def **by** (fastforce simp add: list\_all2\_conv\_all\_nth)

**lemma** mem\_restr\_IntD: mem\_restr (A  $\cap$  B) v  $\Longrightarrow \text{mem\_restr } A v \wedge \text{mem\_restr } B v$

**unfolding** mem\_restr\_def **by** auto

**lemma** mem\_restr\_Un\_iff: mem\_restr (A  $\cup$  B) x  $\longleftrightarrow \text{mem\_restr } A x \vee \text{mem\_restr } B x$

**unfolding** mem\_restr\_def **by** blast

**lemma** mem\_restr\_UNIV [simp]: mem\_restr UNIV x

**unfolding** mem\_restr\_def

**by** (auto simp add: list\_rel\_map intro!: exI[of \_ map the x] list\_rel\_refl)

**lemma** restrict\_mem\_restr[simp]: mem\_restr A x  $\Longrightarrow \text{mem\_restr } A (\text{restrict } V x)$

**unfolding** mem\_restr\_def restrict\_def

**by** (auto simp: list\_all2\_conv\_all\_nth elim!: bexI[rotated])

**definition** lift\_envs :: 'a list set  $\Rightarrow$  'a list set **where**

lift\_envs R = ( $\lambda(a,b). a \# b$ ) ‘ (UNIV  $\times$  R)

**lemma** lift\_envs\_mem\_restr[simp]: mem\_restr A x  $\Longrightarrow \text{mem\_restr } (\text{lift\_envs } A) (a \# x)$

**by** (auto simp: mem\_restr\_def lift\_envs\_def)

**lemma** qtable\_project:

assumes qtable (Suc n) A (mem\_restr (lift\_envs R)) P X

shows qtable n (( $\lambda x. x - \text{Suc } 0$ ) ‘ (A - {0})) (mem\_restr R)

( $\lambda v. \exists x. P ((\text{if } 0 \in A \text{ then Some } x \text{ else None}) \# v)$ ) (tl ‘ X)

(is qtable n ?A (mem\_restr R) ?P ?X)

**proof** ((rule qtableI; (elim exE)?), goal\_cases table left right)

case table

```

with assms show ?case
  unfolding qtable_def by (simp add: table_project)
next
case (left v)
from assms have []  $\notin$  X
  unfolding qtable_def table_def by fastforce
with left(1) obtain x where  $x \# v \in X$ 
  by (metis (no_types, opaque_lifting) image_iff hd_Cons_tl)
with assms show ?case
  by (rule in_qtableE) (auto simp: left(3) split: if_splits)
next
case (right v x)
with assms have (if  $0 \in A$  then Some x else None)  $\# v \in X$ 
  by (elim in_qtableI) auto
then show ?case
  by (auto simp: image_iff elim: bexI[rotated])
qed

```

```

lemma qtable_cong: qtable n A P Q X  $\implies$   $A = B \implies (\bigwedge v. P v \implies Q v \longleftrightarrow Q' v) \implies$  qtable n B P Q' X
  by (auto simp: qtable_def)

```

### 3 Abstract monitors and slicing

#### 3.1 First-order specifications

We abstract from first-order trace specifications by referring only to their semantics. A first-order specification is described by a finite set of free variables and a satisfaction function that defines for every trace the pairs of valuations and time-points for which the specification is satisfied.

```

locale fo_spec =
  fixes
    nfv :: nat and fv :: nat set and
    sat :: 'a trace  $\Rightarrow$  'b list  $\Rightarrow$  nat  $\Rightarrow$  bool
  assumes
    fv_less_nfv:  $x \in fv \implies x < nfv$  and
    sat_fv_cong:  $(\bigwedge x. x \in fv \implies v!x = v!x) \implies sat \sigma v i = sat \sigma v' i$ 
begin

```

```

definition verdicts :: 'a trace  $\Rightarrow$  (nat  $\times$  'b tuple) set where
  verdicts  $\sigma = \{(i, v). wf\_tuple\ nfv\ fv\ v \wedge sat\ \sigma\ (map\ the\ v)\ i\}$ 

```

**end**

We usually employ a monitor to find the *violations* of a specification. That is, the monitor should output the satisfactions of its negation. Moreover, all monitor implementations must work with finite prefixes. We are therefore interested in co-safety properties, which allow us to identify all satisfactions on finite prefixes.

```

locale cosafety_fo_spec = fo_spec +
  assumes cosafety_lr:  $sat\ \sigma\ v\ i \implies \exists \pi. prefix\_of\ \pi\ \sigma \wedge (\forall \sigma'. prefix\_of\ \pi\ \sigma' \longrightarrow sat\ \sigma'\ v\ i)$ 
begin

```

```

lemma cosafety:  $sat\ \sigma\ v\ i \longleftrightarrow (\exists \pi. prefix\_of\ \pi\ \sigma \wedge (\forall \sigma'. prefix\_of\ \pi\ \sigma' \longrightarrow sat\ \sigma'\ v\ i))$ 
  using cosafety_lr by blast

```

**end**

## 3.2 Monitor function

We model monitors abstractly as functions from prefixes to verdict sets. The following locale specifies a minimal set of properties that any reasonable monitor should have.

```

locale monitor = fo_spec +
  fixes M :: 'a prefix  $\Rightarrow$  (nat  $\times$  'b tuple) set
  assumes
    mono_monitor:  $\pi \leq \pi' \Longrightarrow M \pi \subseteq M \pi'$  and
    wf_monitor:  $(i, v) \in M \pi \Longrightarrow wf\_tuple \ nfv \ fv \ v$  and
    sound_monitor:  $(i, v) \in M \pi \Longrightarrow prefix\_of \ \pi \ \sigma \Longrightarrow sat \ \sigma \ (map \ the \ v) \ i$  and
    complete_monitor:  $prefix\_of \ \pi \ \sigma \Longrightarrow wf\_tuple \ nfv \ fv \ v \Longrightarrow$ 
       $(\bigwedge \sigma. prefix\_of \ \pi \ \sigma \Longrightarrow sat \ \sigma \ (map \ the \ v) \ i) \Longrightarrow \exists \pi'. prefix\_of \ \pi' \ \sigma \wedge (i, v) \in M \pi'$ 

```

A monitor for a co-safety specification computes precisely the set of all satisfactions in the limit:

```

abbreviation (in monitor) M_limit  $\sigma \equiv \bigcup \{M \pi \mid \pi. prefix\_of \ \pi \ \sigma\}$ 

```

```

locale cosafety_monitor = cosafety_fo_spec + monitor
begin

```

```

lemma M_limit_eq: M_limit  $\sigma = verdicts \ \sigma$ 

```

```

proof

```

```

  show  $\bigcup \{M \pi \mid \pi. prefix\_of \ \pi \ \sigma\} \subseteq verdicts \ \sigma$ 
  by (auto simp: verdicts_def wf_monitor sound_monitor)

```

```

next

```

```

  show  $\bigcup \{M \pi \mid \pi. prefix\_of \ \pi \ \sigma\} \supseteq verdicts \ \sigma$ 
  unfolding verdicts_def

```

```

proof safe

```

```

  fix i v

```

```

  assume wf_tuple nfv fv v and sat  $\sigma \ (map \ the \ v) \ i$ 

```

```

  then obtain  $\pi$  where prefix_of  $\pi \ \sigma \wedge (\forall \sigma'. prefix\_of \ \pi \ \sigma' \longrightarrow sat \ \sigma' \ (map \ the \ v) \ i)$ 

```

```

  using cosafety_lr by blast

```

```

  with  $\langle wf\_tuple \ nfv \ fv \ v \rangle$  obtain  $\pi'$  where prefix_of  $\pi' \ \sigma \wedge (i, v) \in M \pi'$ 

```

```

  using complete_monitor by blast

```

```

  then show  $(i, v) \in \bigcup \{M \pi \mid \pi. prefix\_of \ \pi \ \sigma\}$ 

```

```

  by blast

```

```

qed

```

```

qed

```

```

end

```

The monitor function  $M$  adds some information over  $sat$ , namely when a verdict is output. One possible behavior is that the monitor outputs its verdicts for one time-point at a time, in increasing order of time-points. Then  $M$  is uniquely defined by a progress function, which returns for every prefix the time-point up to which all verdicts are computed.

```

locale progress = fo_spec __ sat for sat :: 'a trace  $\Rightarrow$  'b list  $\Rightarrow$  nat  $\Rightarrow$  bool +

```

```

  fixes progress :: 'a prefix  $\Rightarrow$  nat

```

```

  assumes

```

```

    progress_mono:  $\pi \leq \pi' \Longrightarrow progress \ \pi \leq progress \ \pi'$  and

```

```

    ex_progress_ge:  $\exists \pi. prefix\_of \ \pi \ \sigma \wedge x \leq progress \ \pi$  and

```

```

    progress_sat_cong:  $prefix\_of \ \pi \ \sigma \Longrightarrow prefix\_of \ \pi \ \sigma' \Longrightarrow i < progress \ \pi \Longrightarrow$ 

```

```

       $sat \ \sigma \ v \ i \longleftrightarrow sat \ \sigma' \ v \ i$ 

```

— The last condition is not necessary to obtain a proper monitor function. However, it corresponds to the intuitive understanding of monitor progress, and it results in a stronger characterisation. In particular, it implies that the specification is co-safety, as we will show below.

```

begin

```

```

definition M :: 'a prefix  $\Rightarrow$  (nat  $\times$  'b tuple) set where

```

```

M π = {(i, v). i < progress π ∧ wf_tuple nfv fv v ∧
  (∀σ. prefix_of π σ → sat σ (map the v) i)}

lemma M_alt: M π = {(i, v). i < progress π ∧ wf_tuple nfv fv v ∧
  (∃σ. prefix_of π σ ∧ sat σ (map the v) i)}
  using ex_prefix_of[of π]
  by (auto simp: M_def cong: progress_sat_cong)

end

sublocale progress ⊆ cosafety_monitor _ _ _ M
proof
  fix i v and σ :: 'a trace
  assume sat σ v i
  moreover obtain π where *: prefix_of π σ i < progress π
    using ex_progress_ge by (auto simp: less_eq_Suc_le)
  ultimately have sat σ' v i if prefix_of π σ' for σ'
    using that by (simp cong: progress_sat_cong)
  with * show ∃π. prefix_of π σ ∧ (∀σ'. prefix_of π σ' → sat σ' v i)
    by blast
next
  fix π π' :: 'a prefix
  assume π ≤ π'
  then show M π ⊆ M π'
    by (auto simp: M_def intro: progress_mono prefix_of_antimono
      elim: order.strict_trans2)
next
  fix i v π and σ :: 'a trace
  assume *: (i, v) ∈ M π
  then show wf_tuple nfv fv v
    by (simp add: M_def)
  assume prefix_of π σ
  with * show sat σ (map the v) i
    by (simp add: M_def)
next
  fix i v π and σ :: 'a trace
  assume *: prefix_of π σ wf_tuple nfv fv v ∧ σ'. prefix_of π σ' ⇒ sat σ' (map the v) i
  show ∃π'. prefix_of π' σ ∧ (i, v) ∈ M π'
  proof (cases i < progress π)
    case True
    with * show ?thesis by (auto simp: M_def)
  next
    case False
    obtain π' where **: prefix_of π' σ ∧ i < progress π'
      using ex_progress_ge by (auto simp: less_eq_Suc_le)
    then have π ≤ π'
      using ⟨prefix_of π σ⟩ prefix_of_imp_linear False progress_mono
      by (blast intro: order.strict_trans2)
    with ** show ?thesis
      by (auto simp: M_def intro: prefix_of_antimono)
  qed
qed

```

### 3.3 Slicing

Sliceable specifications can be evaluated meaningfully on a subset of events.

```

locale abstract_slicer =
  fixes relevant_events :: 'b list set ⇒ 'a set

```

**begin**

**abbreviation**  $\text{slice} :: 'b \text{ list set} \Rightarrow 'a \text{ trace} \Rightarrow 'a \text{ trace}$  **where**  
 $\text{slice } S \equiv \text{map\_}\Gamma (\lambda D. D \cap \text{relevant\_events } S)$

**abbreviation**  $\text{pslice} :: 'b \text{ list set} \Rightarrow 'a \text{ prefix} \Rightarrow 'a \text{ prefix}$  **where**  
 $\text{pslice } S \equiv \text{pmap\_}\Gamma (\lambda D. D \cap \text{relevant\_events } S)$

**lemma**  $\text{prefix\_of\_psliceI}$ :  $\text{prefix\_of } \pi \sigma \Longrightarrow \text{prefix\_of } (\text{pslice } S \pi) (\text{slice } S \sigma)$   
**by** ( $\text{transfer fixing: } S$ ) *auto*

**lemma**  $\text{plen\_pslice[simp]}$ :  $\text{plen } (\text{pslice } S \pi) = \text{plen } \pi$   
**by** ( $\text{transfer fixing: } S$ ) *simp*

**lemma**  $\text{pslice\_pnil[simp]}$ :  $\text{pslice } S \text{pnil} = \text{pnil}$   
**by** ( $\text{transfer fixing: } S$ ) *simp*

**lemma**  $\text{last\_ts\_pslice[simp]}$ :  $\text{last\_ts } (\text{pslice } S \pi) = \text{last\_ts } \pi$   
**by** ( $\text{transfer fixing: } S$ ) (*simp add: last\_map case\_prod\_beta split: list.split*)

**abbreviation**  $\text{verdict\_filter} :: 'b \text{ list set} \Rightarrow (\text{nat} \times 'b \text{ tuple}) \text{ set} \Rightarrow (\text{nat} \times 'b \text{ tuple}) \text{ set}$  **where**  
 $\text{verdict\_filter } S V \equiv \{(i, v) \in V. \text{mem\_restr } S v\}$

**end**

**locale**  $\text{sliceable\_fo\_spec} = \text{fo\_spec } \_ \_ \text{sat} + \text{abstract\_slicer relevant\_events}$   
**for**  $\text{relevant\_events} :: 'b \text{ list set} \Rightarrow 'a \text{ set}$  **and**  $\text{sat} :: 'a \text{ trace} \Rightarrow 'b \text{ list} \Rightarrow \text{nat} \Rightarrow \text{bool} +$   
**assumes**  $\text{sliceable}$ :  $v \in S \Longrightarrow \text{sat } (\text{slice } S \sigma) v i \longleftrightarrow \text{sat } \sigma v i$   
**begin**

**lemma**  $\text{union\_verdicts\_slice}$ :  
**assumes**  $\text{part}$ :  $\bigcup S = \text{UNIV}$   
**shows**  $\bigcup ((\lambda S. \text{verdict\_filter } S (\text{verdicts } (\text{slice } S \sigma))) 'S) = \text{verdicts } \sigma$   
**proof safe**

**fix**  $S i$  **and**  $v :: 'b \text{ tuple}$   
**assume**  $(i, v) \in \text{verdicts } (\text{slice } S \sigma)$   
**then have**  $\text{tuple}$ :  $\text{wf\_tuple } \text{nfv } \text{fv } v$  **and**  $\text{sat } (\text{slice } S \sigma) (\text{map the } v) i$   
**by** (*auto simp: verdicts\_def*)  
**assume**  $\text{mem\_restr } S v$   
**then obtain**  $v'$  **where**  $v' \in S$  **and**  $1$ :  $\forall i \in \text{fv}. v ! i = \text{Some } (v' ! i)$   
**using**  $\text{tuple}$  **by** (*auto simp: fv\_less\_nfv elim: mem\_restrE*)  
**then have**  $\text{sat } (\text{slice } S \sigma) v' i$   
**using**  $\langle \text{sat } (\text{slice } S \sigma) (\text{map the } v) i \rangle \text{tuple}$   
**by** (*auto simp: wf\_tuple\_length fv\_less\_nfv cong: sat\_fv\_cong*)  
**then have**  $\text{sat } \sigma v' i$   
**using**  $\text{sliceable}[OF \langle v' \in S \rangle]$  **by** *simp*  
**then have**  $\text{sat } \sigma (\text{map the } v) i$   
**using**  $\text{tuple } 1$   
**by** (*auto simp: wf\_tuple\_length fv\_less\_nfv cong: sat\_fv\_cong*)  
**then show**  $(i, v) \in \text{verdicts } \sigma$   
**using**  $\text{tuple}$  **by** (*simp add: verdicts\_def*)

**next**

**fix**  $i$  **and**  $v :: 'b \text{ tuple}$   
**assume**  $(i, v) \in \text{verdicts } \sigma$   
**then have**  $\text{tuple}$ :  $\text{wf\_tuple } \text{nfv } \text{fv } v$  **and**  $\text{sat } \sigma (\text{map the } v) i$   
**by** (*auto simp: verdicts\_def*)  
**from part obtain**  $S$  **where**  $S \in S$  **and**  $\text{map the } v \in S$   
**by** *blast*

```

then have mem_restr S v
  using mem_restrI[of map the v S nfv fv] tuple
  by (auto simp: wf_tuple_def fv_less_nfv)
moreover have sat (slice S  $\sigma$ ) (map the v) i
  using  $\langle \text{sat } \sigma \text{ (map the v) } i \rangle$  sliceable[OF  $\langle \text{map the v } \in S \rangle$ ] by simp
then have  $(i, v) \in \text{verdicts } (\text{slice } S \ \sigma)$ 
  using tuple by (simp add: verdicts_def)
ultimately show  $(i, v) \in (\bigcup S \in \mathcal{S}. \text{verdict\_filter } S \ (\text{verdicts } (\text{slice } S \ \sigma)))$ 
  using  $\langle S \in \mathcal{S} \rangle$  by blast
qed

```

**end**

We define a similar notion for monitors. It is potentially stronger because the time-point at which verdicts are output must not change.

```

locale sliceable_monitor = monitor  $\_ \_ \text{sat } M + \text{abstract\_slicer } \text{relevant\_events}$ 
  for relevant_events :: 'b list set  $\Rightarrow$  'a set and sat :: 'a trace  $\Rightarrow$  'b list  $\Rightarrow$  nat  $\Rightarrow$  bool and M +
  assumes sliceable_M: mem_restr S v  $\Longrightarrow$   $(i, v) \in M \ (\text{pslice } S \ \pi) \longleftrightarrow (i, v) \in M \ \pi$ 
begin

```

```

lemma union_M_pslice:
  assumes part:  $\bigcup \mathcal{S} = \text{UNIV}$ 
  shows  $\bigcup ((\lambda S. \text{verdict\_filter } S \ (M \ (\text{pslice } S \ \pi))) \ \mathcal{S}) = M \ \pi$ 
proof safe

```

```

  fix S i and v :: 'b tuple
  assume mem_restr S v and  $(i, v) \in M \ (\text{pslice } S \ \pi)$ 
  then show  $(i, v) \in M \ \pi$  using sliceable_M by blast
next
  fix i and v :: 'b tuple
  assume  $(i, v) \in M \ \pi$ 
  then have tuple: wf_tuple nfv fv v
    by (rule wf_monitor)
  from part obtain S where  $S \in \mathcal{S}$  and map the v  $\in S$ 
    by blast
  then have mem_restr S v
    using mem_restrI[of map the v S nfv fv] tuple
    by (auto simp: wf_tuple_def fv_less_nfv)
  then have  $(i, v) \in M \ (\text{pslice } S \ \pi)$ 
    using  $\langle (i, v) \in M \ \pi \rangle$  sliceable_M by blast
  then show  $(i, v) \in (\bigcup S \in \mathcal{S}. \text{verdict\_filter } S \ (M \ (\text{pslice } S \ \pi)))$ 
    using  $\langle S \in \mathcal{S} \rangle$   $\langle \text{mem\_restr } S \ v \rangle$  by blast
qed

```

**end**

If the specification is sliceable and the monitor's progress depends only on time-stamps, then also the monitor itself is sliceable.

```

locale timed_progress = progress +
  assumes progress_time_conv: pts  $\pi = \text{pts } \pi' \Longrightarrow \text{progress } \pi = \text{progress } \pi'$ 

```

```

locale sliceable_timed_progress = sliceable_fo_spec + timed_progress
begin

```

```

lemma progress_pslice[simp]: progress (pslice S  $\pi$ ) = progress  $\pi$ 
  by (simp cong: progress_time_conv)

```

**end**

```

sublocale sliceable_timed_progress ⊆ sliceable_monitor _ _ _ _ M
proof
  fix S :: 'a list set and v i π
  assume *: mem_restr S v
  show (i, v) ∈ M (pslice S π) ⟷ (i, v) ∈ M π (is ?L ⟷ ?R)
  proof
    assume ?L
    with * show ?R
      by (auto 0 4 simp: M_def wf_tuple_def elim!: mem_restrE
          box_equals[OF sliceable_sat_fv_cong sat_fv_cong, THEN iffD1, rotated -1]
          intro: prefix_of_psliceI dest: fv_less_nfv spec[of _ slice S _])
    next
      assume ?R
      with * show ?L
        by (auto 0 4 simp: M_alt_wf_tuple_def elim!: mem_restrE
            box_equals[OF sliceable_sat_fv_cong sat_fv_cong, THEN iffD2, rotated -1]
            intro: exI[of _ slice S _] prefix_of_psliceI dest: fv_less_nfv)
  qed
qed

```

## 4 Intervals

```

typedef  $\mathcal{I} = \{(i :: nat, j :: enat). i \leq j\}$ 
  by (intro exI[of _ (0, ∞)]) auto

```

```

setup_lifting type_definition  $\mathcal{I}$ 

```

```

instantiation  $\mathcal{I} :: equal$  begin
lift_definition equal_ $\mathcal{I} :: \mathcal{I} \Rightarrow \mathcal{I} \Rightarrow bool$  is (=) .
instance by standard (transfer, auto)
end

```

```

instantiation  $\mathcal{I} :: linorder$  begin
lift_definition less_eq_ $\mathcal{I} :: \mathcal{I} \Rightarrow \mathcal{I} \Rightarrow bool$  is (≤) .
lift_definition less_ $\mathcal{I} :: \mathcal{I} \Rightarrow \mathcal{I} \Rightarrow bool$  is (<) .
instance by standard (transfer, auto)+
end

```

```

lift_definition all ::  $\mathcal{I}$  is (0, ∞) by simp
lift_definition left ::  $\mathcal{I} \Rightarrow nat$  is fst .
lift_definition right ::  $\mathcal{I} \Rightarrow enat$  is snd .
lift_definition point ::  $nat \Rightarrow \mathcal{I}$  is λn. (n, enat n) by simp
lift_definition init ::  $nat \Rightarrow \mathcal{I}$  is λn. (0, enat n) by auto
abbreviation mem where mem n I ≡ (left I ≤ n ∧ n ≤ right I)
lift_definition subtract ::  $nat \Rightarrow \mathcal{I} \Rightarrow \mathcal{I}$  is
  λn (i, j). (i - n, j - enat n) by (auto simp: diff_enat_def split: enat.splits)
lift_definition add ::  $nat \Rightarrow \mathcal{I} \Rightarrow \mathcal{I}$  is
  λn (a, b). (a, b + enat n) by (auto simp add: add_increasing2)

```

```

lemma left_right: left I ≤ right I
  by transfer auto

```

```

lemma point_simps[simp]:
  left (point n) = n
  right (point n) = n
  by (transfer; auto)+

```

**lemma** *init\_simps*[simp]:

*left* (*init* *n*) = 0  
*right* (*init* *n*) = *n*  
**by** (*transfer*; *auto*)+

**lemma** *subtract\_simps*[simp]:

*left* (*subtract* *n* *I*) = *left* *I* - *n*  
*right* (*subtract* *n* *I*) = *right* *I* - *n*  
*subtract* 0 *I* = *I*  
*subtract* *x* (*point* *y*) = *point* (*y* - *x*)  
**by** (*transfer*; *auto*)+

**definition** *shifted* ::  $\mathcal{I} \Rightarrow \mathcal{I}$  **set where**

*shifted* *I* = ( $\lambda n. \text{subtract } n \text{ } I$ ) ‘ {0 .. (case *right* *I* of  $\infty \Rightarrow \text{left } I \mid \text{enat } n \Rightarrow n$ )}

**lemma** *subtract\_too\_much*:  $i > (\text{case } \text{right } I \text{ of } \infty \Rightarrow \text{left } I \mid \text{enat } n \Rightarrow n) \implies$

*subtract* *i* *I* = *subtract* (*case* *right* *I* of  $\infty \Rightarrow \text{left } I \mid \text{enat } n \Rightarrow n$ ) *I*  
**by** *transfer* (*auto split*: *enat.splits*)

**lemma** *subtract\_shifted*: *subtract* *n* *I*  $\in$  *shifted* *I*

**by** (*cases*  $n \leq (\text{case } \text{right } I \text{ of } \infty \Rightarrow \text{left } I \mid \text{enat } n \Rightarrow n)$ )  
(*auto simp*: *shifted\_def subtract\_too\_much*)

**lemma** *finite\_shifted*: *finite* (*shifted* *I*)

**unfolding** *shifted\_def* **by** *auto*

**definition** *interval* ::  $\text{nat} \Rightarrow \text{enat} \Rightarrow \mathcal{I}$  **where**

*interval* *l* *r* = (*if*  $l \leq r$  then *Abs*  $\mathcal{I}$  (*l*, *r*) else *undefined*)

**lemma** [*code abstract*]: *Rep*  $\mathcal{I}$  (*interval* *l* *r*) = (*if*  $l \leq r$  then (*l*, *r*) else *Rep*  $\mathcal{I}$  *undefined*)

**unfolding** *interval\_def* **using** *Abs*  $\mathcal{I}$  *inverse* **by** *simp*

## 5 Metric first-order temporal logic

**context** *begin*

### 5.1 Formulas and satisfiability

**qualified type\_synonym** *name* = *string*

**qualified type\_synonym** *'a event* = (*name*  $\times$  *'a list*)

**qualified type\_synonym** *'a database* = *'a event set*

**qualified type\_synonym** *'a prefix* = (*name*  $\times$  *'a list*) *prefix*

**qualified type\_synonym** *'a trace* = (*name*  $\times$  *'a list*) *trace*

**qualified type\_synonym** *'a env* = *'a list*

**qualified datatype** *'a trm* = *Var* *nat* | *is\_Const*: *Const* *'a*

**qualified primrec** *fvi\_trm* ::  $\text{nat} \Rightarrow 'a \text{ trm} \Rightarrow \text{nat set}$  **where**

*fvi\_trm* *b* (*Var* *x*) = (*if*  $b \leq x$  then {*x* - *b*} else {})

| *fvi\_trm* *b* (*Const*  $\_$ ) = {}

**abbreviation** *fv\_trm*  $\equiv$  *fvi\_trm* 0

**qualified primrec** *eval\_trm* :: *'a env*  $\Rightarrow$  *'a trm*  $\Rightarrow$  *'a* **where**

*eval\_trm* *v* (*Var* *x*) = *v* ! *x*

|  $eval\_trm\ v\ (Const\ x) = x$

**lemma**  $eval\_trm\_cong$ :  $\forall x \in fv\_trm\ t.\ v!\ x = v'!\ x \implies eval\_trm\ v\ t = eval\_trm\ v'\ t$   
**by** ( $cases\ t$ )  $simp\_all$

**qualified datatype** ( $discs\_sels$ )  $'a\ formula = Pred\ name\ 'a\ trm\ list \mid Eq\ 'a\ trm\ 'a\ trm$   
|  $Neg\ 'a\ formula \mid Or\ 'a\ formula\ 'a\ formula \mid Exists\ 'a\ formula$   
|  $Prev\ \mathcal{I}\ 'a\ formula \mid Next\ \mathcal{I}\ 'a\ formula$   
|  $Since\ 'a\ formula\ \mathcal{I}\ 'a\ formula \mid Until\ 'a\ formula\ \mathcal{I}\ 'a\ formula$

**qualified primrec**  $fvi :: nat \Rightarrow 'a\ formula \Rightarrow nat\ set$  **where**

$fvi\ b\ (Pred\ r\ ts) = (\bigcup t \in set\ ts.\ fvi\_trm\ b\ t)$   
|  $fvi\ b\ (Eq\ t1\ t2) = fvi\_trm\ b\ t1 \cup fvi\_trm\ b\ t2$   
|  $fvi\ b\ (Neg\ \varphi) = fvi\ b\ \varphi$   
|  $fvi\ b\ (Or\ \varphi\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$   
|  $fvi\ b\ (Exists\ \varphi) = fvi\ (Suc\ b)\ \varphi$   
|  $fvi\ b\ (Prev\ I\ \varphi) = fvi\ b\ \varphi$   
|  $fvi\ b\ (Next\ I\ \varphi) = fvi\ b\ \varphi$   
|  $fvi\ b\ (Since\ \varphi\ I\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$   
|  $fvi\ b\ (Until\ \varphi\ I\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$

**abbreviation**  $fv \equiv fvi\ 0$

**lemma**  $finite\_fvi\_trm[simp]$ :  $finite\ (fvi\_trm\ b\ t)$   
**by** ( $cases\ t$ )  $simp\_all$

**lemma**  $finite\_fvi[simp]$ :  $finite\ (fvi\ b\ \varphi)$   
**by** ( $induction\ \varphi\ arbitrary: b$ )  $simp\_all$

**lemma**  $fvi\_trm\_Suc$ :  $x \in fvi\_trm\ (Suc\ b)\ t \longleftrightarrow Suc\ x \in fvi\_trm\ b\ t$   
**by** ( $cases\ t$ )  $auto$

**lemma**  $fvi\_Suc$ :  $x \in fvi\ (Suc\ b)\ \varphi \longleftrightarrow Suc\ x \in fvi\ b\ \varphi$   
**by** ( $induction\ \varphi\ arbitrary: b$ ) ( $simp\_all\ add: fvi\_trm\_Suc$ )

**lemma**  $fvi\_Suc\_bound$ :  
**assumes**  $\forall i \in fvi\ (Suc\ b)\ \varphi.\ i < n$   
**shows**  $\forall i \in fvi\ b\ \varphi.\ i < Suc\ n$

**proof**

**fix**  $i$

**assume**  $i \in fvi\ b\ \varphi$

**with**  $assms$  **show**  $i < Suc\ n$  **by** ( $cases\ i$ ) ( $simp\_all\ add: fvi\_Suc$ )

**qed**

**qualified definition**  $nfv :: 'a\ formula \Rightarrow nat$  **where**  
 $nfv\ \varphi = Max\ (insert\ 0\ (Suc\ 'fv\ \varphi))$

**qualified definition**  $envs :: 'a\ formula \Rightarrow 'a\ env\ set$  **where**  
 $envs\ \varphi = \{v.\ length\ v = nfv\ \varphi\}$

**lemma**  $nfv\_simps[simp]$ :

$nfv\ (Neg\ \varphi) = nfv\ \varphi$   
 $nfv\ (Or\ \varphi\ \psi) = max\ (nfv\ \varphi)\ (nfv\ \psi)$   
 $nfv\ (Prev\ I\ \varphi) = nfv\ \varphi$   
 $nfv\ (Next\ I\ \varphi) = nfv\ \varphi$

$nfv\ (Since\ \varphi\ I\ \psi) = max\ (nfv\ \varphi)\ (nfv\ \psi)$

$nfv\ (Until\ \varphi\ I\ \psi) = max\ (nfv\ \varphi)\ (nfv\ \psi)$

**unfolding**  $nfv\_def$  **by** ( $simp\_all\ add: image\_Un\ Max\_Un[symmetric]$ )

**lemma** *fvi\_less\_nfv*:  $\forall i \in \text{fv } \varphi. i < \text{nfv } \varphi$   
**unfolding** *nfv\_def*  
**by** (*auto simp add: Max\_gr\_iff intro: max.strict\_coboundedI2*)

**qualified primrec** *future\_reach* :: 'a formula  $\Rightarrow$  enat where

*future\_reach* (Pred  $\_ \_$ ) = 0  
| *future\_reach* (Eq  $\_ \_$ ) = 0  
| *future\_reach* (Neg  $\varphi$ ) = *future\_reach*  $\varphi$   
| *future\_reach* (Or  $\varphi \psi$ ) = max (*future\_reach*  $\varphi$ ) (*future\_reach*  $\psi$ )  
| *future\_reach* (Exists  $\varphi$ ) = *future\_reach*  $\varphi$   
| *future\_reach* (Prev  $I \varphi$ ) = *future\_reach*  $\varphi$  - left  $I$   
| *future\_reach* (Next  $I \varphi$ ) = *future\_reach*  $\varphi$  + right  $I$  + 1  
| *future\_reach* (Since  $\varphi I \psi$ ) = max (*future\_reach*  $\varphi$ ) (*future\_reach*  $\psi$  - left  $I$ )  
| *future\_reach* (Until  $\varphi I \psi$ ) = max (*future\_reach*  $\varphi$ ) (*future\_reach*  $\psi$ ) + right  $I$  + 1

**qualified primrec** *sat* :: 'a trace  $\Rightarrow$  'a env  $\Rightarrow$  nat  $\Rightarrow$  'a formula  $\Rightarrow$  bool where

*sat*  $\sigma v i$  (Pred  $r ts$ ) = (( $r$ , map (*eval\_trm*  $v$ )  $ts$ )  $\in \Gamma \sigma i$ )  
| *sat*  $\sigma v i$  (Eq  $t1 t2$ ) = (*eval\_trm*  $v t1$  = *eval\_trm*  $v t2$ )  
| *sat*  $\sigma v i$  (Neg  $\varphi$ ) = ( $\neg$  *sat*  $\sigma v i \varphi$ )  
| *sat*  $\sigma v i$  (Or  $\varphi \psi$ ) = (*sat*  $\sigma v i \varphi \vee$  *sat*  $\sigma v i \psi$ )  
| *sat*  $\sigma v i$  (Exists  $\varphi$ ) = ( $\exists z. \text{sat } \sigma (z \# v) i \varphi$ )  
| *sat*  $\sigma v i$  (Prev  $I \varphi$ ) = (case  $i$  of 0  $\Rightarrow$  False | Suc  $j \Rightarrow$  mem ( $\tau \sigma i - \tau \sigma j$ )  $I \wedge$  *sat*  $\sigma v j \varphi$ )  
| *sat*  $\sigma v i$  (Next  $I \varphi$ ) = (mem ( $\tau \sigma$  (Suc  $i$ ) -  $\tau \sigma i$ )  $I \wedge$  *sat*  $\sigma v$  (Suc  $i$ )  $\varphi$ )  
| *sat*  $\sigma v i$  (Since  $\varphi I \psi$ ) = ( $\exists j \leq i. \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge \text{sat } \sigma v j \psi \wedge (\forall k \in \{j .. i\}. \text{sat } \sigma v k \varphi)$ )  
| *sat*  $\sigma v i$  (Until  $\varphi I \psi$ ) = ( $\exists j \geq i. \text{mem } (\tau \sigma j - \tau \sigma i) I \wedge \text{sat } \sigma v j \psi \wedge (\forall k \in \{i .. j\}. \text{sat } \sigma v k \varphi)$ )

**lemma** *sat\_Until\_rec*: *sat*  $\sigma v i$  (Until  $\varphi I \psi$ )  $\longleftrightarrow$

mem 0  $I \wedge$  *sat*  $\sigma v i \psi \vee$   
 $(\Delta \sigma (i + 1) \leq$  right  $I \wedge$  *sat*  $\sigma v i \varphi \wedge$  *sat*  $\sigma v (i + 1)$  (Until  $\varphi$  (subtract ( $\Delta \sigma (i + 1)$ )  $I$ )  $\psi$ )  
(is ?L  $\longleftrightarrow$  ?R)

**proof** (*rule iffI*; (*elim disjE conjE*)?)

**assume** ?L

**then obtain**  $j$  **where**  $j: i \leq j$  mem ( $\tau \sigma j - \tau \sigma i$ )  $I$  *sat*  $\sigma v j \psi \forall k \in \{i .. j\}. \text{sat } \sigma v k \varphi$

**by** *auto*

**then show** ?R

**proof** (*cases*  $i = j$ )

**case** False

**with**  $j(1,2)$  **have**  $\Delta \sigma (i + 1) \leq$  right  $I$

**by** (*auto elim: order\_trans[rotated] simp: diff\_le\_mono*)

**moreover from** False  $j(1,4)$  **have** *sat*  $\sigma v i \varphi$  **by** *auto*

**moreover from** False  $j$  **have** *sat*  $\sigma v (i + 1)$  (Until  $\varphi$  (subtract ( $\Delta \sigma (i + 1)$ )  $I$ )  $\psi$ )

**by** (*cases* right  $I$ ) (*auto simp: le\_diff\_conv le\_diff\_conv2 intro!: exI[of \_  $j$ ]*)

**ultimately show** ?thesis **by** *blast*

**qed** *simp*

**next**

**assume**  $\Delta: \Delta \sigma (i + 1) \leq$  right  $I$  **and now:** *sat*  $\sigma v i \varphi$  **and**

*next:* *sat*  $\sigma v (i + 1)$  (Until  $\varphi$  (subtract ( $\Delta \sigma (i + 1)$ )  $I$ )  $\psi$ )

**from** *next* **obtain**  $j$  **where**  $j: i + 1 \leq j$  mem ( $\tau \sigma j - \tau \sigma (i + 1)$ ) ((subtract ( $\Delta \sigma (i + 1)$ )  $I$ ))

*sat*  $\sigma v j \psi \forall k \in \{i + 1 .. j\}. \text{sat } \sigma v k \varphi$

**by** *auto*

**from**  $\Delta j(1,2)$  **have** mem ( $\tau \sigma j - \tau \sigma i$ )  $I$

**by** (*cases* right  $I$ ) (*auto simp: le\_diff\_conv2*)

**with** *now*  $j(1,3,4)$  **show** ?L **by** (*auto simp: le\_eq\_less\_or\_eq[of  $i$ ] intro!: exI[of \_  $j$ ]*)

**qed** *auto*

**lemma** *sat\_Since\_rec*:  $\text{sat } \sigma v i \text{ (Since } \varphi I \psi) \longleftrightarrow$   
 $\text{mem } 0 I \wedge \text{sat } \sigma v i \psi \vee$   
 $(i > 0 \wedge \Delta \sigma i \leq \text{right } I \wedge \text{sat } \sigma v i \varphi \wedge \text{sat } \sigma v (i - 1) \text{ (Since } \varphi \text{ (subtract } (\Delta \sigma i) I) \psi))$   
 $(\text{is } ?L \longleftrightarrow ?R)$

**proof** (*rule iffI*; (*elim disjE conjE*)?)  
**assume** *?L*  
**then obtain** *j* **where**  $j: j \leq i \text{ mem } (\tau \sigma i - \tau \sigma j) I \text{ sat } \sigma v j \psi \forall k \in \{j <.. i\}. \text{sat } \sigma v k \varphi$   
**by** *auto*  
**then show** *?R*  
**proof** (*cases i = j*)  
**case** *False*  
**with**  $j(1)$  **obtain** *k* **where** [*simp*]:  $i = k + 1$   
**by** (*cases i*) *auto*  
**with**  $j(1,2)$  *False* **have**  $\Delta \sigma i \leq \text{right } I$   
**by** (*auto elim: order\_trans[rotated] simp: diff\_le\_mono2 le\_Suc\_eq*)  
**moreover from** *False j(1,4)* **have**  $\text{sat } \sigma v i \varphi$  **by** *auto*  
**moreover from** *False j* **have**  $\text{sat } \sigma v (i - 1) \text{ (Since } \varphi \text{ (subtract } (\Delta \sigma i) I) \psi)$   
**by** (*cases right I*) (*auto simp: le\_diff\_conv le\_diff\_conv2 intro!: exI[of \_ j]*)  
**ultimately show** *?thesis* **by** *auto*  
**qed** *simp*

**next**  
**assume**  $i: 0 < i$  **and**  $\Delta: \Delta \sigma i \leq \text{right } I$  **and now:**  $\text{sat } \sigma v i \varphi$  **and**  
 $\text{prev: sat } \sigma v (i - 1) \text{ (Since } \varphi \text{ (subtract } (\Delta \sigma i) I) \psi)$   
**from** *prev* **obtain** *j* **where**  $j: j \leq i - 1 \text{ mem } (\tau \sigma (i - 1) - \tau \sigma j) ((\text{subtract } (\Delta \sigma i) I))$   
 $\text{sat } \sigma v j \psi \forall k \in \{j <.. i - 1\}. \text{sat } \sigma v k \varphi$   
**by** *auto*  
**from**  $\Delta i j(1,2)$  **have**  $\text{mem } (\tau \sigma i - \tau \sigma j) I$   
**by** (*cases right I*) (*auto simp: le\_diff\_conv2*)  
**with** *now i j(1,3,4)* **show** *?L* **by** (*auto simp: le\_Suc\_eq gr0\_conv\_Suc intro!: exI[of \_ j]*)  
**qed** *auto*

**lemma** *sat\_Since\_0*:  $\text{sat } \sigma v 0 \text{ (Since } \varphi I \psi) \longleftrightarrow \text{mem } 0 I \wedge \text{sat } \sigma v 0 \psi$   
**by** *auto*

**lemma** *sat\_Since\_point*:  $\text{sat } \sigma v i \text{ (Since } \varphi I \psi) \implies$   
 $(\bigwedge j. j \leq i \implies \text{mem } (\tau \sigma i - \tau \sigma j) I \implies \text{sat } \sigma v i \text{ (Since } \varphi \text{ (point } (\tau \sigma i - \tau \sigma j)) \psi) \implies P) \implies P$   
**by** (*auto intro: diff\_le\_self*)

**lemma** *sat\_Since\_pointD*:  $\text{sat } \sigma v i \text{ (Since } \varphi \text{ (point } t) \psi) \implies \text{mem } t I \implies \text{sat } \sigma v i \text{ (Since } \varphi I \psi)$   
**by** *auto*

**lemma** *eval\_trm\_fvi\_cong*:  $\forall x \in \text{fv\_trm } t. v!x = v'!x \implies \text{eval\_trm } v t = \text{eval\_trm } v' t$   
**by** (*cases t*) *simp\_all*

**lemma** *sat\_fvi\_cong*:  $\forall x \in \text{fv } \varphi. v!x = v'!x \implies \text{sat } \sigma v i \varphi = \text{sat } \sigma v' i \varphi$   
**proof** (*induct*  $\varphi$  *arbitrary: v v' i*)  
**case** (*Pred n ts*)  
**show** *?case* **by** (*simp cong: map\_cong eval\_trm\_fvi\_cong[OF Pred[simplified, THEN bspec]]*)  
**next**  
**case** (*Eq x1 x2*)  
**then show** *?case* **unfolding** *fvi.simps sat.simps* **by** (*metis UnCI eval\_trm\_fvi\_cong*)  
**next**  
**case** (*Exists*  $\varphi$ )  
**then show** *?case* **unfolding** *sat.simps* **by** (*intro iff\_exI*) (*simp add: fvi\_Suc nth\_Cons'*)  
**qed** (*auto*  $8\ 0$  *simp add: nth\_Cons' split: nat.splits intro!: iff\_exI*)

## 5.2 Defined connectives

**qualified definition**  $And\ \varphi\ \psi = Neg\ (Or\ (Neg\ \varphi)\ (Neg\ \psi))$

**lemma**  $fvi\_And$ :  $fvi\ b\ (And\ \varphi\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$   
**unfolding**  $And\_def$  **by**  $simp$

**lemma**  $nfv\_And[simp]$ :  $nfv\ (And\ \varphi\ \psi) = max\ (nfv\ \varphi)\ (nfv\ \psi)$   
**unfolding**  $nfv\_def$  **by** ( $simp\ add$ :  $fvi\_And\ image\_Un\ Max\_Un[symmetric]$ )

**lemma**  $future\_reach\_And$ :  $future\_reach\ (And\ \varphi\ \psi) = max\ (future\_reach\ \varphi)\ (future\_reach\ \psi)$   
**unfolding**  $And\_def$  **by**  $simp$

**lemma**  $sat\_And$ :  $sat\ \sigma\ v\ i\ (And\ \varphi\ \psi) = (sat\ \sigma\ v\ i\ \varphi \wedge sat\ \sigma\ v\ i\ \psi)$   
**unfolding**  $And\_def$  **by**  $simp$

**qualified definition**  $And\_Not\ \varphi\ \psi = Neg\ (Or\ (Neg\ \varphi)\ \psi)$

**lemma**  $fvi\_And\_Not$ :  $fvi\ b\ (And\_Not\ \varphi\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$   
**unfolding**  $And\_Not\_def$  **by**  $simp$

**lemma**  $nfv\_And\_Not[simp]$ :  $nfv\ (And\_Not\ \varphi\ \psi) = max\ (nfv\ \varphi)\ (nfv\ \psi)$   
**unfolding**  $nfv\_def$  **by** ( $simp\ add$ :  $fvi\_And\_Not\ image\_Un\ Max\_Un[symmetric]$ )

**lemma**  $future\_reach\_And\_Not$ :  $future\_reach\ (And\_Not\ \varphi\ \psi) = max\ (future\_reach\ \varphi)\ (future\_reach\ \psi)$   
**unfolding**  $And\_Not\_def$  **by**  $simp$

**lemma**  $sat\_And\_Not$ :  $sat\ \sigma\ v\ i\ (And\_Not\ \varphi\ \psi) = (sat\ \sigma\ v\ i\ \varphi \wedge \neg sat\ \sigma\ v\ i\ \psi)$   
**unfolding**  $And\_Not\_def$  **by**  $simp$

## 5.3 Safe formulas

**fun**  $safe\_formula :: 'a\ MFOTL.formula \Rightarrow bool$  **where**

$safe\_formula\ (MFOTL.Eq\ t1\ t2) = (MFOTL.is\_Const\ t1 \vee MFOTL.is\_Const\ t2)$   
 $| safe\_formula\ (MFOTL.Neg\ (MFOTL.Eq\ (MFOTL.Const\ x)\ (MFOTL.Const\ y))) = True$   
 $| safe\_formula\ (MFOTL.Neg\ (MFOTL.Eq\ (MFOTL.Var\ x)\ (MFOTL.Var\ y))) = (x = y)$   
 $| safe\_formula\ (MFOTL.Pred\ e\ ts) = True$   
 $| safe\_formula\ (MFOTL.Neg\ (MFOTL.Or\ (MFOTL.Neg\ \varphi)\ \psi)) = (safe\_formula\ \varphi \wedge$   
 $(safe\_formula\ \psi \wedge MFOTL.fv\ \psi \subseteq MFOTL.fv\ \varphi \vee (case\ \psi\ of\ MFOTL.Neg\ \psi' \Rightarrow safe\_formula\ \psi' |$   
 $\_ \Rightarrow False)))$   
 $| safe\_formula\ (MFOTL.Or\ \varphi\ \psi) = (MFOTL.fv\ \psi = MFOTL.fv\ \varphi \wedge safe\_formula\ \varphi \wedge safe\_formula$   
 $\psi)$   
 $| safe\_formula\ (MFOTL.Exists\ \varphi) = (safe\_formula\ \varphi)$   
 $| safe\_formula\ (MFOTL.Prev\ I\ \varphi) = (safe\_formula\ \varphi)$   
 $| safe\_formula\ (MFOTL.Next\ I\ \varphi) = (safe\_formula\ \varphi)$   
 $| safe\_formula\ (MFOTL.Since\ \varphi\ I\ \psi) = (MFOTL.fv\ \varphi \subseteq MFOTL.fv\ \psi \wedge$   
 $(safe\_formula\ \varphi \vee (case\ \varphi\ of\ MFOTL.Neg\ \varphi' \Rightarrow safe\_formula\ \varphi' | \_ \Rightarrow False)) \wedge safe\_formula\ \psi)$   
 $| safe\_formula\ (MFOTL.Until\ \varphi\ I\ \psi) = (MFOTL.fv\ \varphi \subseteq MFOTL.fv\ \psi \wedge$   
 $(safe\_formula\ \varphi \vee (case\ \varphi\ of\ MFOTL.Neg\ \varphi' \Rightarrow safe\_formula\ \varphi' | \_ \Rightarrow False)) \wedge safe\_formula\ \psi)$   
 $| safe\_formula\ \_ = False$

**lemma**  $disjE\_Not2$ :  $P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (\neg P \Longrightarrow Q \Longrightarrow R) \Longrightarrow R$   
**by**  $blast$

**lemma**  $safe\_formula\_induct[consumes\ 1]$ :  
**assumes**  $safe\_formula\ \varphi$   
**and**  $\bigwedge t1\ t2. MFOTL.is\_Const\ t1 \Longrightarrow P\ (MFOTL.Eq\ t1\ t2)$   
**and**  $\bigwedge t1\ t2. MFOTL.is\_Const\ t2 \Longrightarrow P\ (MFOTL.Eq\ t1\ t2)$

```

and  $\bigwedge x y. P (MFOTL.Neg (MFOTL.Eq (MFOTL.Const x) (MFOTL.Const y)))$ 
and  $\bigwedge x y. x = y \implies P (MFOTL.Neg (MFOTL.Eq (MFOTL.Var x) (MFOTL.Var y)))$ 
and  $\bigwedge e ts. P (MFOTL.Pred e ts)$ 
and  $\bigwedge \varphi \psi. \neg (safe\_formula (MFOTL.Neg \psi) \wedge MFOTL.fv \psi \subseteq MFOTL.fv \varphi) \implies P \varphi \implies P \psi$ 
 $\implies P (MFOTL.And \varphi \psi)$ 
and  $\bigwedge \varphi \psi. safe\_formula \psi \implies MFOTL.fv \psi \subseteq MFOTL.fv \varphi \implies P \varphi \implies P \psi \implies P (MFOTL.And\_Not$ 
 $\varphi \psi)$ 
and  $\bigwedge \varphi \psi. MFOTL.fv \psi = MFOTL.fv \varphi \implies P \varphi \implies P \psi \implies P (MFOTL.Or \varphi \psi)$ 
and  $\bigwedge \varphi. P \varphi \implies P (MFOTL.Exists \varphi)$ 
and  $\bigwedge I \varphi. P \varphi \implies P (MFOTL.Prev I \varphi)$ 
and  $\bigwedge I \varphi. P \varphi \implies P (MFOTL.Next I \varphi)$ 
and  $\bigwedge \varphi I \psi. MFOTL.fv \varphi \subseteq MFOTL.fv \psi \implies safe\_formula \varphi \implies P \varphi \implies P \psi \implies P (MFOTL.Since$ 
 $\varphi I \psi)$ 
and  $\bigwedge \varphi I \psi. MFOTL.fv (MFOTL.Neg \varphi) \subseteq MFOTL.fv \psi \implies$ 
 $\neg safe\_formula (MFOTL.Neg \varphi) \implies P \varphi \implies P \psi \implies P (MFOTL.Since (MFOTL.Neg \varphi) I \psi)$ 
and  $\bigwedge \varphi I \psi. MFOTL.fv \varphi \subseteq MFOTL.fv \psi \implies safe\_formula \varphi \implies P \varphi \implies P \psi \implies P (MFOTL.Until$ 
 $\varphi I \psi)$ 
and  $\bigwedge \varphi I \psi. MFOTL.fv (MFOTL.Neg \varphi) \subseteq MFOTL.fv \psi \implies$ 
 $\neg safe\_formula (MFOTL.Neg \varphi) \implies P \varphi \implies P \psi \implies P (MFOTL.Until (MFOTL.Neg \varphi) I \psi)$ 
shows  $P \varphi$ 
using  $assms(1)$ 
proof (induction rule: safe_formula.induct)
case ( $5 \varphi \psi$ )
then show  $?case$ 
by ( $cases \psi$ )
  ( $auto 0 3 elim!: disjE\_Not2 intro: assms[unfolded MFOTL.And\_def MFOTL.And\_Not\_def]$ )
next
case ( $10 \varphi I \psi$ )
then show  $?case$ 
by ( $cases \varphi$ ) ( $auto 0 3 elim!: disjE\_Not2 intro: assms$ )
next
case ( $11 \varphi I \psi$ )
then show  $?case$ 
by ( $cases \varphi$ ) ( $auto 0 3 elim!: disjE\_Not2 intro: assms$ )
qed ( $auto intro: assms$ )

```

## 5.4 Slicing traces

**qualified primrec**  $matches :: 'a env \Rightarrow 'a formula \Rightarrow name \times 'a list \Rightarrow bool$  **where**

```

 $matches v (Pred r ts) e = (r = fst e \wedge map (eval\_trm v) ts = snd e)$ 
 $| matches v (Eq \_ \_) e = False$ 
 $| matches v (Neg \varphi) e = matches v \varphi e$ 
 $| matches v (Or \varphi \psi) e = (matches v \varphi e \vee matches v \psi e)$ 
 $| matches v (Exists \varphi) e = (\exists z. matches (z \# v) \varphi e)$ 
 $| matches v (Prev I \varphi) e = matches v \varphi e$ 
 $| matches v (Next I \varphi) e = matches v \varphi e$ 
 $| matches v (Since \varphi I \psi) e = (matches v \varphi e \vee matches v \psi e)$ 
 $| matches v (Until \varphi I \psi) e = (matches v \varphi e \vee matches v \psi e)$ 

```

**lemma**  $matches\_fvi\_cong: \forall x \in fv \varphi. v!x = v'!x \implies matches v \varphi e = matches v' \varphi e$

**proof** (*induct  $\varphi$  arbitrary:  $v v'$* )

**case** ( $Pred n ts$ )

**show**  $?case$  **by** ( $simp cong: map\_cong eval\_trm\_fvi\_cong[OF Pred[simplified, THEN bspec]]$ )

**next**

**case** ( $Exists \varphi$ )

**then show**  $?case$  **unfolding**  $matches.simps$  **by** ( $intro iff\_exI$ ) ( $simp add: fvi\_Suc nth\_Cons'$ )

**qed** ( $auto 5 0 simp add: nth\_Cons'$ )

**abbreviation** *relevant\_events* **where** *relevant\_events*  $\varphi S \equiv \{e. S \cap \{v. \text{matches } v \varphi e\} \neq \{\}\}$

**lemma** *sat\_slice\_strong*: *relevant\_events*  $\varphi S \subseteq E \implies v \in S \implies$

$\text{sat } \sigma v i \varphi \longleftrightarrow \text{sat } (\text{map\_}\Gamma (\lambda D. D \cap E) \sigma) v i \varphi$

**proof** (*induction*  $\varphi$  *arbitrary*:  $v S i$ )

**case** (*Pred*  $r ts$ )

**then show** *?case* **by** (*auto simp: subset\_eq*)

**next**

**case** (*Eq*  $t1 t2$ )

**show** *?case*

**unfolding** *sat.simps*

**by** *simp*

**next**

**case** (*Neg*  $\varphi$ )

**then show** *?case* **by** *simp*

**next**

**case** (*Or*  $\varphi \psi$ )

**show** *?case* **using** *Or.IH[of S] Or.prem*s

**by** (*auto simp: Collect\_disj\_eq Int\_Un\_distrib subset\_iff*)

**next**

**case** (*Exists*  $\varphi$ )

**have**  $\text{sat } \sigma (z \# v) i \varphi = \text{sat } (\text{map\_}\Gamma (\lambda D. D \cap E) \sigma) (z \# v) i \varphi$  **for**  $z$

**using** *Exists.prem*s **by** (*auto intro!: Exists.IH[of {z # v | v. v \in S}]*)

**then show** *?case* **by** *simp*

**next**

**case** (*Prev*  $I \varphi$ )

**then show** *?case* **by** (*auto cong: nat.case\_cong*)

**next**

**case** (*Next*  $I \varphi$ )

**then show** *?case* **by** *simp*

**next**

**case** (*Since*  $\varphi I \psi$ )

**show** *?case* **using** *Since.IH[of S] Since.prem*s

**by** (*auto simp: Collect\_disj\_eq Int\_Un\_distrib subset\_iff*)

**next**

**case** (*Until*  $\varphi I \psi$ )

**show** *?case* **using** *Until.IH[of S] Until.prem*s

**by** (*auto simp: Collect\_disj\_eq Int\_Un\_distrib subset\_iff*)

**qed**

**end**

**interpretation** *MFOTL\_slicer*: *abstract\_slicer relevant\_events*  $\varphi$  **for**  $\varphi$  .

**lemma** *sat\_slice\_iff*:

**assumes**  $v \in S$

**shows**  $\text{MFOTL.sat } \sigma v i \varphi \longleftrightarrow \text{MFOTL.sat } (\text{MFOTL\_slicer.slice } \varphi S \sigma) v i \varphi$

**by** (*rule sat\_slice\_strong[of S, OF subset\_refl assms]*)

**lemma** *slice\_replace\_prefix*:

*prefix\_of* (*MFOTL\_slicer.pslice*  $\varphi R \pi$ )  $\sigma \implies$

$\text{MFOTL\_slicer.slice } \varphi R (\text{replace\_prefix } \pi \sigma) = \text{MFOTL\_slicer.slice } \varphi R \sigma$

**by** (*rule map\_\Gamma\_replace\_prefix*) *auto*

## 6 Monitor implementation

### 6.1 Monitorable formulas

**definition**  $mmonitorable\ \varphi \iff safe\_formula\ \varphi \wedge MFOTL.future\_reach\ \varphi \neq \infty$

```

fun mmonitorable_exec :: 'a MFOTL.formula  $\Rightarrow$  bool where
  mmonitorable_exec (MFOTL.Eq t1 t2) = (MFOTL.is_Const t1  $\vee$  MFOTL.is_Const t2)
| mmonitorable_exec (MFOTL.Neg (MFOTL.Eq (MFOTL.Const x) (MFOTL.Const y))) = True
| mmonitorable_exec (MFOTL.Neg (MFOTL.Eq (MFOTL.Var x) (MFOTL.Var y))) = (x = y)
| mmonitorable_exec (MFOTL.Pred e ts) = True
| mmonitorable_exec (MFOTL.Neg (MFOTL.Or (MFOTL.Neg  $\varphi$ )  $\psi$ )) = (mmonitorable_exec  $\varphi$   $\wedge$ 
  (mmonitorable_exec  $\psi$   $\wedge$  MFOTL.fv  $\psi$   $\subseteq$  MFOTL.fv  $\varphi$   $\vee$  (case  $\psi$  of MFOTL.Neg  $\psi'$   $\Rightarrow$  mmoni-
  torable_exec  $\psi'$  | _  $\Rightarrow$  False)))
| mmonitorable_exec (MFOTL.Or  $\varphi$   $\psi$ ) = (MFOTL.fv  $\psi$  = MFOTL.fv  $\varphi$   $\wedge$  mmonitorable_exec  $\varphi$   $\wedge$ 
  mmonitorable_exec  $\psi$ )
| mmonitorable_exec (MFOTL.Exists  $\varphi$ ) = (mmonitorable_exec  $\varphi$ )
| mmonitorable_exec (MFOTL.Prev I  $\varphi$ ) = (mmonitorable_exec  $\varphi$ )
| mmonitorable_exec (MFOTL.Next I  $\varphi$ ) = (mmonitorable_exec  $\varphi$   $\wedge$  right I  $\neq \infty$ )
| mmonitorable_exec (MFOTL.Since  $\varphi$  I  $\psi$ ) = (MFOTL.fv  $\varphi$   $\subseteq$  MFOTL.fv  $\psi$   $\wedge$ 
  (mmonitorable_exec  $\varphi$   $\vee$  (case  $\varphi$  of MFOTL.Neg  $\varphi'$   $\Rightarrow$  mmonitorable_exec  $\varphi'$  | _  $\Rightarrow$  False))  $\wedge$ 
  mmonitorable_exec  $\psi$ )
| mmonitorable_exec (MFOTL.Until  $\varphi$  I  $\psi$ ) = (MFOTL.fv  $\varphi$   $\subseteq$  MFOTL.fv  $\psi$   $\wedge$  right I  $\neq \infty$   $\wedge$ 
  (mmonitorable_exec  $\varphi$   $\vee$  (case  $\varphi$  of MFOTL.Neg  $\varphi'$   $\Rightarrow$  mmonitorable_exec  $\varphi'$  | _  $\Rightarrow$  False))  $\wedge$ 
  mmonitorable_exec  $\psi$ )
| mmonitorable_exec _ = False

```

**lemma** plus\_eq\_enat\_iff:  $a + b = enat\ i \iff (\exists j\ k. a = enat\ j \wedge b = enat\ k \wedge j + k = i)$   
**by** (cases a; cases b) auto

**lemma** minus\_eq\_enat\_iff:  $a - enat\ k = enat\ i \iff (\exists j. a = enat\ j \wedge j - k = i)$   
**by** (cases a) auto

**lemma** safe\_formula\_mmonitorable\_exec:  $safe\_formula\ \varphi \implies MFOTL.future\_reach\ \varphi \neq \infty \implies mmoni-$   
 $torable\_exec\ \varphi$

**proof** (induct  $\varphi$  rule: safe\_formula.induct)

**case** (5  $\varphi\ \psi$ )

**then show** ?case

**unfolding** safe\_formula.simps future\_reach.simps mmonitorable\_exec.simps

**by** (fastforce split: formula.splits)

**next**

**case** (6  $\varphi\ \psi$ )

**then show** ?case

**unfolding** safe\_formula.simps future\_reach.simps mmonitorable\_exec.simps

**by** (fastforce split: formula.splits)

**next**

**case** (10  $\varphi\ I\ \psi$ )

**then show** ?case

**unfolding** safe\_formula.simps future\_reach.simps mmonitorable\_exec.simps

**by** (fastforce split: formula.splits)

**next**

**case** (11  $\varphi\ I\ \psi$ )

**then show** ?case

**unfolding** safe\_formula.simps future\_reach.simps mmonitorable\_exec.simps

**by** (fastforce simp: plus\_eq\_enat\_iff split: formula.splits)

**qed** (auto simp add: plus\_eq\_enat\_iff minus\_eq\_enat\_iff)

**lemma** mmonitorable\_exec\_mmonitorable:  $mmonitorable\_exec\ \varphi \implies mmonitorable\ \varphi$

**proof** (induct  $\varphi$  rule: mmonitorable\_exec.induct)

```

case (5  $\varphi$   $\psi$ )
then show ?case
  unfolding mmonitorable_def mmonitorable_exec.simps safe_formula.simps
  by (fastforce split: formula.splits)
next
case (10  $\varphi$   $I$   $\psi$ )
then show ?case
  unfolding mmonitorable_def mmonitorable_exec.simps safe_formula.simps
  by (fastforce split: formula.splits)
next
case (11  $\varphi$   $I$   $\psi$ )
then show ?case
  unfolding mmonitorable_def mmonitorable_exec.simps safe_formula.simps
  by (fastforce simp: one_enat_def split: formula.splits)
qed (auto simp add: mmonitorable_def one_enat_def)

lemma monitorable_formula_code[code]: mmonitorable  $\varphi$  = mmonitorable_exec  $\varphi$ 
  using mmonitorable_exec_mmonitorable safe_formula_mmonitorable_exec mmonitorable_def
  by blast

```

## 6.2 The executable monitor

```

type_synonym ts = nat

```

```

type_synonym 'a mbuf2 = 'a table list  $\times$  'a table list
type_synonym 'a msaux = (ts  $\times$  'a table) list
type_synonym 'a mauaux = (ts  $\times$  'a table  $\times$  'a table) list

```

```

datatype 'a mformula =
  MRel 'a table
| MPred MFOTL.name 'a MFOTL.trm list
| MAnd 'a mformula bool 'a mformula 'a mbuf2
| MOr 'a mformula 'a mformula 'a mbuf2
| MExists 'a mformula
| MPrev  $\mathcal{I}$  'a mformula bool 'a table list ts list
| MNext  $\mathcal{I}$  'a mformula bool ts list
| MSince bool 'a mformula  $\mathcal{I}$  'a mformula 'a mbuf2 ts list 'a msaux
| MUntil bool 'a mformula  $\mathcal{I}$  'a mformula 'a mbuf2 ts list 'a mauaux

```

```

record 'a mstate =
  mstate_i :: nat
  mstate_m :: 'a mformula
  mstate_n :: nat

```

```

fun eq_rel :: nat  $\Rightarrow$  'a MFOTL.trm  $\Rightarrow$  'a MFOTL.trm  $\Rightarrow$  'a table where
  eq_rel n (MFOTL.Const x) (MFOTL.Const y) = (if x = y then unit_table n else empty_table)
| eq_rel n (MFOTL.Var x) (MFOTL.Const y) = singleton_table n x y
| eq_rel n (MFOTL.Const x) (MFOTL.Var y) = singleton_table n y x
| eq_rel n (MFOTL.Var x) (MFOTL.Var y) = undefined

```

```

fun neq_rel :: nat  $\Rightarrow$  'a MFOTL.trm  $\Rightarrow$  'a MFOTL.trm  $\Rightarrow$  'a table where
  neq_rel n (MFOTL.Const x) (MFOTL.Const y) = (if x = y then empty_table else unit_table n)
| neq_rel n (MFOTL.Var x) (MFOTL.Var y) = (if x = y then empty_table else undefined)
| neq_rel _ _ _ = undefined

```

```

fun minit0 :: nat  $\Rightarrow$  'a MFOTL.formula  $\Rightarrow$  'a mformula where
  minit0 n (MFOTL.Neg  $\varphi$ ) = (case  $\varphi$  of
    MFOTL.Eq t1 t2  $\Rightarrow$  MRel (neq_rel n t1 t2)

```

```

| MFOTL.Or (MFOTL.Neg  $\varphi$ )  $\psi$   $\Rightarrow$  (if safe_formula  $\psi \wedge$  MFOTL.fv  $\psi \subseteq$  MFOTL.fv  $\varphi$ 
  then MAnd (minit0 n  $\varphi$ ) False (minit0 n  $\psi$ ) ([], [])
  else (case  $\psi$  of MFOTL.Neg  $\psi \Rightarrow$  MAnd (minit0 n  $\varphi$ ) True (minit0 n  $\psi$ ) ([], []) | _  $\Rightarrow$  undefined))
| _  $\Rightarrow$  undefined)
| minit0 n (MFOTL.Eq t1 t2) = MRel (eq_rel n t1 t2)
| minit0 n (MFOTL.Pred e ts) = MPred e ts
| minit0 n (MFOTL.Or  $\varphi$   $\psi$ ) = MOr (minit0 n  $\varphi$ ) (minit0 n  $\psi$ ) ([], [])
| minit0 n (MFOTL.Exists  $\varphi$ ) = MExists (minit0 (Suc n)  $\varphi$ )
| minit0 n (MFOTL.Prev I  $\varphi$ ) = MPrev I (minit0 n  $\varphi$ ) True [] []
| minit0 n (MFOTL.Next I  $\varphi$ ) = MNext I (minit0 n  $\varphi$ ) True []
| minit0 n (MFOTL.Since  $\varphi$  I  $\psi$ ) = (if safe_formula  $\varphi$ 
  then MSince True (minit0 n  $\varphi$ ) I (minit0 n  $\psi$ ) ([], []) [] []
  else (case  $\varphi$  of
    MFOTL.Neg  $\varphi \Rightarrow$  MSince False (minit0 n  $\varphi$ ) I (minit0 n  $\psi$ ) ([], []) [] []
    | _  $\Rightarrow$  undefined))
| minit0 n (MFOTL.Until  $\varphi$  I  $\psi$ ) = (if safe_formula  $\varphi$ 
  then MUntil True (minit0 n  $\varphi$ ) I (minit0 n  $\psi$ ) ([], []) [] []
  else (case  $\varphi$  of
    MFOTL.Neg  $\varphi \Rightarrow$  MUntil False (minit0 n  $\varphi$ ) I (minit0 n  $\psi$ ) ([], []) [] []
    | _  $\Rightarrow$  undefined))

```

**definition** minit :: 'a MFOTL.formula  $\Rightarrow$  'a mstate **where**

minit  $\varphi$  = (let n = MFOTL.nfv  $\varphi$  in (mstate\_i = 0, mstate\_m = minit0 n  $\varphi$ , mstate\_n = n))

**fun** mprev\_next ::  $\mathcal{I} \Rightarrow$  'a table list  $\Rightarrow$  ts list  $\Rightarrow$  'a table list  $\times$  'a table list  $\times$  ts list **where**

```

mprev_next I [] ts = ([], [], ts)
| mprev_next I xs [] = ([], xs, [])
| mprev_next I xs [t] = ([], xs, [t])
| mprev_next I (x # xs) (t # t' # ts) = (let (ys, zs) = mprev_next I xs (t' # ts)
  in ((if mem (t' - t) I then x else empty_table) # ys, zs))

```

**fun** mbuf2\_add :: 'a table list  $\Rightarrow$  'a table list  $\Rightarrow$  'a mbuf2  $\Rightarrow$  'a mbuf2 **where**

mbuf2\_add xs' ys' (xs, ys) = (xs @ xs', ys @ ys')

**fun** mbuf2\_take :: ('a table  $\Rightarrow$  'a table  $\Rightarrow$  'b)  $\Rightarrow$  'a mbuf2  $\Rightarrow$  'b list  $\times$  'a mbuf2 **where**

```

mbuf2_take f (x # xs, y # ys) = (let (zs, buf) = mbuf2_take f (xs, ys) in (f x y # zs, buf))
| mbuf2_take f (xs, ys) = ([], (xs, ys))

```

**fun** mbuf2t\_take :: ('a table  $\Rightarrow$  'a table  $\Rightarrow$  ts  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'b  $\Rightarrow$

```

'a mbuf2  $\Rightarrow$  ts list  $\Rightarrow$  'b  $\times$  'a mbuf2  $\times$  ts list where
mbuf2t_take f z (x # xs, y # ys) (t # ts) = mbuf2t_take f (f x y t z) (xs, ys) ts
| mbuf2t_take f z (xs, ys) ts = (z, (xs, ys), ts)

```

**fun** match :: 'a MFOTL.trm list  $\Rightarrow$  'a list  $\Rightarrow$  (nat  $\rightarrow$  'a) option **where**

```

match [] [] = Some Map.empty
| match (MFOTL.Const x # ts) (y # ys) = (if x = y then match ts ys else None)
| match (MFOTL.Var x # ts) (y # ys) = (case match ts ys of
  None  $\Rightarrow$  None
  | Some f  $\Rightarrow$  (case f x of
    None  $\Rightarrow$  Some (f(x  $\mapsto$  y))
    | Some z  $\Rightarrow$  if y = z then Some f else None))
| match _ _ = None

```

**definition** update\_since ::  $\mathcal{I} \Rightarrow$  bool  $\Rightarrow$  'a table  $\Rightarrow$  'a table  $\Rightarrow$  ts  $\Rightarrow$

'a msaux  $\Rightarrow$  'a table  $\times$  'a msaux **where**

```

update_since I pos rel1 rel2 nt aux =
  (let aux = (case [(t, join rel pos rel1). (t, rel)  $\leftarrow$  aux, nt - t  $\leq$  right I] of
    []  $\Rightarrow$  [(nt, rel2)]

```

$| x \# aux' \Rightarrow (if \text{fst } x = nt \text{ then } (\text{fst } x, \text{snd } x \cup \text{rel2}) \# aux' \text{ else } (nt, \text{rel2}) \# x \# aux')$   
 $in (\text{foldr } (\cup) [\text{rel}. (t, \text{rel}) \leftarrow aux, \text{left } I \leq nt - t] \{\}, aux)$

**definition**  $\text{update\_until} :: \mathcal{I} \Rightarrow \text{bool} \Rightarrow 'a \text{ table} \Rightarrow 'a \text{ table} \Rightarrow ts \Rightarrow 'a \text{ muaux} \Rightarrow 'a \text{ muaux}$  **where**  
 $\text{update\_until } I \text{ pos } \text{rel1 } \text{rel2 } nt \text{ aux} =$   
 $(\text{map } (\lambda x. \text{case } x \text{ of } (t, a1, a2) \Rightarrow (t, \text{if } \text{pos} \text{ then } \text{join } a1 \text{ True } \text{rel1} \text{ else } a1 \cup \text{rel1},$   
 $\text{if } \text{mem } (nt - t) \text{ I then } a2 \cup \text{join } \text{rel2 } \text{pos } a1 \text{ else } a2)) \text{ aux}) @$   
 $[(nt, \text{rel1}, \text{if } \text{left } I = 0 \text{ then } \text{rel2} \text{ else } \text{empty\_table})]$

**fun**  $\text{eval\_until} :: \mathcal{I} \Rightarrow ts \Rightarrow 'a \text{ muaux} \Rightarrow 'a \text{ table list} \times 'a \text{ muaux}$  **where**  
 $\text{eval\_until } I \text{ nt } [] = ([], [])$   
 $| \text{eval\_until } I \text{ nt } ((t, a1, a2) \# aux) = (\text{if } t + \text{right } I < nt \text{ then}$   
 $(\text{let } (xs, aux) = \text{eval\_until } I \text{ nt } aux \text{ in } (a2 \# xs, aux)) \text{ else } ([], (t, a1, a2) \# aux))$

**primrec**  $\text{meval} :: \text{nat} \Rightarrow ts \Rightarrow 'a \text{ MFOTL.database} \Rightarrow 'a \text{ mformula} \Rightarrow 'a \text{ table list} \times 'a \text{ mformula}$  **where**  
 $\text{meval } n \text{ t db } (M\text{Rel } \text{rel}) = ([\text{rel}], M\text{Rel } \text{rel})$   
 $| \text{meval } n \text{ t db } (M\text{Pred } e \text{ ts}) = ((\lambda f. \text{tabulate } f \ 0 \ n) ' \text{Option.these}$   
 $(\text{match } ts \text{ ' } (\bigcup (e', x) \in \text{db}. \text{if } e = e' \text{ then } \{x\} \text{ else } \{\}))], M\text{Pred } e \text{ ts})$   
 $| \text{meval } n \text{ t db } (M\text{And } \varphi \text{ pos } \psi \text{ buf}) =$   
 $(\text{let } (xs, \varphi) = \text{meval } n \text{ t db } \varphi; (ys, \psi) = \text{meval } n \text{ t db } \psi;$   
 $(zs, \text{buf}) = \text{mbuf2\_take } (\lambda r1 \ r2. \text{join } r1 \text{ pos } r2) (\text{mbuf2\_add } xs \text{ ys } \text{buf})$   
 $\text{in } (zs, M\text{And } \varphi \text{ pos } \psi \text{ buf}))$   
 $| \text{meval } n \text{ t db } (M\text{Or } \varphi \text{ } \psi \text{ buf}) =$   
 $(\text{let } (xs, \varphi) = \text{meval } n \text{ t db } \varphi; (ys, \psi) = \text{meval } n \text{ t db } \psi;$   
 $(zs, \text{buf}) = \text{mbuf2\_take } (\lambda r1 \ r2. r1 \cup r2) (\text{mbuf2\_add } xs \text{ ys } \text{buf})$   
 $\text{in } (zs, M\text{Or } \varphi \text{ } \psi \text{ buf}))$   
 $| \text{meval } n \text{ t db } (M\text{Exists } \varphi) =$   
 $(\text{let } (xs, \varphi) = \text{meval } (\text{Suc } n) \text{ t db } \varphi \text{ in } (\text{map } (\lambda r. \text{tl } ' r) \text{ xs}, M\text{Exists } \varphi))$   
 $| \text{meval } n \text{ t db } (M\text{Prev } I \varphi \text{ first } \text{buf } \text{nts}) =$   
 $(\text{let } (xs, \varphi) = \text{meval } n \text{ t db } \varphi;$   
 $(zs, \text{buf}, \text{nts}) = \text{mprev\_next } I (\text{buf } @ \text{xs}) (\text{nts } @ [t])$   
 $\text{in } (\text{if } \text{first} \text{ then } \text{empty\_table} \# \text{zs} \text{ else } \text{zs}, M\text{Prev } I \varphi \text{ False } \text{buf } \text{nts}))$   
 $| \text{meval } n \text{ t db } (M\text{Next } I \varphi \text{ first } \text{nts}) =$   
 $(\text{let } (xs, \varphi) = \text{meval } n \text{ t db } \varphi;$   
 $(\text{xs}, \text{first}) = (\text{case } (\text{xs}, \text{first}) \text{ of } (\_ \# \text{xs}, \text{True}) \Rightarrow (\text{xs}, \text{False}) \mid a \Rightarrow a);$   
 $(\text{zs}, \_, \text{nts}) = \text{mprev\_next } I \text{ xs } (\text{nts } @ [t])$   
 $\text{in } (zs, M\text{Next } I \varphi \text{ first } \text{nts}))$   
 $| \text{meval } n \text{ t db } (M\text{Since } \text{pos } \varphi \text{ I } \psi \text{ buf } \text{nts } \text{aux}) =$   
 $(\text{let } (xs, \varphi) = \text{meval } n \text{ t db } \varphi; (ys, \psi) = \text{meval } n \text{ t db } \psi;$   
 $((\text{zs}, \text{aux}), \text{buf}, \text{nts}) = \text{mbuf2t\_take } (\lambda r1 \ r2 \ t (\text{zs}, \text{aux}).$   
 $\text{let } (z, \text{aux}) = \text{update\_since } I \text{ pos } r1 \ r2 \ t \text{ aux}$   
 $\text{in } (\text{zs } @ [z], \text{aux})) ([], \text{aux}) (\text{mbuf2\_add } xs \text{ ys } \text{buf}) (\text{nts } @ [t])$   
 $\text{in } (zs, M\text{Since } \text{pos } \varphi \text{ I } \psi \text{ buf } \text{nts } \text{aux}))$   
 $| \text{meval } n \text{ t db } (M\text{Until } \text{pos } \varphi \text{ I } \psi \text{ buf } \text{nts } \text{aux}) =$   
 $(\text{let } (xs, \varphi) = \text{meval } n \text{ t db } \varphi; (ys, \psi) = \text{meval } n \text{ t db } \psi;$   
 $(\text{aux}, \text{buf}, \text{nts}) = \text{mbuf2t\_take } (\text{update\_until } I \text{ pos}) \text{ aux } (\text{mbuf2\_add } xs \text{ ys } \text{buf}) (\text{nts } @ [t]);$   
 $(\text{zs}, \text{aux}) = \text{eval\_until } I (\text{case } \text{nts} \text{ of } [] \Rightarrow t \mid nt \# \_ \Rightarrow nt) \text{ aux}$   
 $\text{in } (zs, M\text{Until } \text{pos } \varphi \text{ I } \psi \text{ buf } \text{nts } \text{aux}))$

**definition**  $\text{mstep} :: 'a \text{ MFOTL.database} \times ts \Rightarrow 'a \text{ mstate} \Rightarrow (\text{nat} \times 'a \text{ tuple}) \text{ set} \times 'a \text{ mstate}$  **where**  
 $\text{mstep } \text{tdb } \text{st} =$   
 $(\text{let } (xs, m) = \text{meval } (\text{mstate\_n } \text{st}) (\text{snd } \text{tdb}) (\text{fst } \text{tdb}) (\text{mstate\_m } \text{st})$   
 $\text{in } (\bigcup (\text{set } (\text{map } (\lambda(i, X). (\lambda v. (i, v)) ' X) (\text{List.enumerate } (\text{mstate\_i } \text{st}) \text{xs}))),$   
 $(\text{mstate\_i} = \text{mstate\_i } \text{st} + \text{length } \text{xs}, \text{mstate\_m} = m, \text{mstate\_n} = \text{mstate\_n } \text{st})))$

**lemma**  $\text{mstep\_alt}$ :  $\text{mstep } \text{tdb } \text{st} =$   
 $(\text{let } (xs, m) = \text{meval } (\text{mstate\_n } \text{st}) (\text{snd } \text{tdb}) (\text{fst } \text{tdb}) (\text{mstate\_m } \text{st})$   
 $\text{in } (\bigcup (i, X) \in \text{set } (\text{List.enumerate } (\text{mstate\_i } \text{st}) \text{xs}). \bigcup v \in X. \{(i, v)\},$

```

    (mstate_i = mstate_i st + length xs, mstate_m = m, mstate_n = mstate_n st)))
  unfolding mstep_def
  by (auto split: prod.split)

```

### 6.3 Progress

```

primrec progress :: 'a MFOTL.trace  $\Rightarrow$  'a MFOTL.formula  $\Rightarrow$  nat  $\Rightarrow$  nat where
  progress  $\sigma$  (MFOTL.Pred e ts) j = j
| progress  $\sigma$  (MFOTL.Eq t1 t2) j = j
| progress  $\sigma$  (MFOTL.Neg  $\varphi$ ) j = progress  $\sigma$   $\varphi$  j
| progress  $\sigma$  (MFOTL.Or  $\varphi$   $\psi$ ) j = min (progress  $\sigma$   $\varphi$  j) (progress  $\sigma$   $\psi$  j)
| progress  $\sigma$  (MFOTL.Exists  $\varphi$ ) j = progress  $\sigma$   $\varphi$  j
| progress  $\sigma$  (MFOTL.Prev I  $\varphi$ ) j = (if j = 0 then 0 else min (Suc (progress  $\sigma$   $\varphi$  j)) j)
| progress  $\sigma$  (MFOTL.Next I  $\varphi$ ) j = progress  $\sigma$   $\varphi$  j - 1
| progress  $\sigma$  (MFOTL.Since  $\varphi$  I  $\psi$ ) j = min (progress  $\sigma$   $\varphi$  j) (progress  $\sigma$   $\psi$  j)
| progress  $\sigma$  (MFOTL.Until  $\varphi$  I  $\psi$ ) j =
  Inf {i.  $\forall k. k < j \wedge k \leq \min$  (progress  $\sigma$   $\varphi$  j) (progress  $\sigma$   $\psi$  j)  $\longrightarrow$   $\tau \sigma i + \text{right } I \geq \tau \sigma k$ }

```

```

lemma progress_And[simp]: progress  $\sigma$  (MFOTL.And  $\varphi$   $\psi$ ) j = min (progress  $\sigma$   $\varphi$  j) (progress  $\sigma$   $\psi$  j)
  unfolding MFOTL.And_def by simp

```

```

lemma progress_And_Not[simp]: progress  $\sigma$  (MFOTL.And_Not  $\varphi$   $\psi$ ) j = min (progress  $\sigma$   $\varphi$  j) (progress
 $\sigma$   $\psi$  j)
  unfolding MFOTL.And_Not_def by simp

```

```

lemma progress_mono: j  $\leq$  j'  $\implies$  progress  $\sigma$   $\varphi$  j  $\leq$  progress  $\sigma$   $\varphi$  j'

```

```

proof (induction  $\varphi$ )
  case (Until  $\varphi$  I  $\psi$ )
  then show ?case
    by (cases right I)
      (auto dest: trans_le_add1[OF  $\tau$ _mono] intro!: cInf_superset_mono)
qed auto

```

```

lemma progress_le: progress  $\sigma$   $\varphi$  j  $\leq$  j

```

```

proof (induction  $\varphi$ )
  case (Until  $\varphi$  I  $\psi$ )
  then show ?case
    by (cases right I)
      (auto intro: trans_le_add1[OF  $\tau$ _mono] intro!: cInf_lower)
qed auto

```

```

lemma progress_0[simp]: progress  $\sigma$   $\varphi$  0 = 0
  using progress_le by auto

```

```

lemma progress_ge: MFOTL.future_reach  $\varphi \neq \infty \implies \exists j. i \leq$  progress  $\sigma$   $\varphi$  j

```

```

proof (induction  $\varphi$  arbitrary: i)
  case (Pred e ts)
  then show ?case by auto
next
  case (Eq t1 t2)
  then show ?case by auto
next
  case (Neg  $\varphi$ )
  then show ?case by simp
next
  case (Or  $\varphi$ 1  $\varphi$ 2)
  from Or.prem1 have MFOTL.future_reach  $\varphi$ 1  $\neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi$ 1) (auto)

```

```

moreover from Or.prems have MFOTL.future_reach  $\varphi 2 \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi 2$ ) (auto)
ultimately obtain  $j1\ j2$  where  $i \leq \text{progress } \sigma\ \varphi 1\ j1$  and  $i \leq \text{progress } \sigma\ \varphi 2\ j2$ 
  using Or.IH[of i] by blast
then have  $i \leq \text{progress } \sigma$  (MFOTL.Or  $\varphi 1\ \varphi 2$ ) (max j1 j2)
  by (cases j1 ≤ j2) (auto elim!: order.trans[OF _ progress_mono])
then show ?case by blast
next
case (Exists  $\varphi$ )
then show ?case by simp
next
case (Prev I  $\varphi$ )
from Prev.prems have MFOTL.future_reach  $\varphi \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi$ ) (auto)
then obtain  $j$  where  $i \leq \text{progress } \sigma\ \varphi\ j$ 
  using Prev.IH[of i] by blast
then show ?case by (auto intro!: exI[of _ j] elim!: order.trans[OF _ progress_le])
next
case (Next I  $\varphi$ )
from Next.prems have MFOTL.future_reach  $\varphi \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi$ ) (auto)
then obtain  $j$  where Suc i  $\leq \text{progress } \sigma\ \varphi\ j$ 
  using Next.IH[of Suc i] by blast
then show ?case by (auto intro!: exI[of _ j])
next
case (Since  $\varphi 1\ I\ \varphi 2$ )
from Since.prems have MFOTL.future_reach  $\varphi 1 \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi 1$ ) (auto)
moreover from Since.prems have MFOTL.future_reach  $\varphi 2 \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi 2$ ) (auto)
ultimately obtain  $j1\ j2$  where  $i \leq \text{progress } \sigma\ \varphi 1\ j1$  and  $i \leq \text{progress } \sigma\ \varphi 2\ j2$ 
  using Since.IH[of i] by blast
then have  $i \leq \text{progress } \sigma$  (MFOTL.Since  $\varphi 1\ I\ \varphi 2$ ) (max j1 j2)
  by (cases j1 ≤ j2) (auto elim!: order.trans[OF _ progress_mono])
then show ?case by blast
next
case (Until  $\varphi 1\ I\ \varphi 2$ )
from Until.prems obtain  $b$  where [simp]: right I = enat b
  by (cases right I) (auto)
obtain  $i'$  where  $i < i'$  and  $\tau\ \sigma\ i + b + 1 \leq \tau\ \sigma\ i'$ 
  using ex_le_τ[where x=τ σ i + b + 1] by (auto simp add: less_eq_Suc_le)
then have  $1: \tau\ \sigma\ i + b < \tau\ \sigma\ i'$  by simp
from Until.prems have MFOTL.future_reach  $\varphi 1 \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi 1$ ) (auto)
moreover from Until.prems have MFOTL.future_reach  $\varphi 2 \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi 2$ ) (auto)
ultimately obtain  $j1\ j2$  where Suc i'  $\leq \text{progress } \sigma\ \varphi 1\ j1$  and Suc i'  $\leq \text{progress } \sigma\ \varphi 2\ j2$ 
  using Until.IH[of Suc i'] by blast
then have  $i \leq \text{progress } \sigma$  (MFOTL.Until  $\varphi 1\ I\ \varphi 2$ ) (max j1 j2)
  unfolding progress.simps
proof (intro cInf_greatest, goal_cases nonempty greatest)
  case nonempty
  then show ?case
    by (auto simp: trans_le_add1[OF τ_mono] intro!: exI[of _ max j1 j2])
next
case (greatest x)
with  $\langle i < i' \rangle 1$  show ?case
  by (cases j1 ≤ j2)

```

```

(auto dest!: spec[of _ i] simp: max_absorb1 max_absorb2 less_eq_Suc_le
 elim: order.trans[OF _ progress_le] order.trans[OF _ progress_mono, rotated]
 dest!: not_le_imp_less[THEN less_imp_le] intro!: less_τD[THEN less_imp_le, of σ i x])
qed
then show ?case by blast
qed

lemma cInf_restrict_nat:
  fixes x :: nat
  assumes x ∈ A
  shows Inf A = Inf {y ∈ A. y ≤ x}
  using assms by (auto intro!: antisym intro: cInf_greatest cInf_lower Inf_nat_def1)

lemma progress_time_conv:
  assumes ∀ i < j. τ σ i = τ σ' i
  shows progress σ φ j = progress σ' φ j
proof (induction φ)
  case (Until φ1 I φ2)
  have *: i ≤ j - 1 ↔ i < j if j ≠ 0 for i
    using that by auto
  with Until show ?case
proof (cases right I)
  case (enat b)
  then show ?thesis
proof (cases j)
  case (Suc n)
  with enat * Until show ?thesis
  using assms τ_mono[THEN trans_le_add1]
  by (auto 6 0
    intro!: box_equals[OF arg_cong[where f=Inf]
      cInf_restrict_nat[symmetric, where x=n] cInf_restrict_nat[symmetric, where x=n]])
  qed simp
  qed simp
qed simp_all

lemma Inf_UNIV_nat: (Inf UNIV :: nat) = 0
  by (simp add: cInf_eq_minimum)

lemma progress_prefix_conv:
  assumes prefix_of π σ and prefix_of π σ'
  shows progress σ φ (plen π) = progress σ' φ (plen π)
  using assms by (auto intro: progress_time_conv τ_prefix_conv)

lemma sat_prefix_conv:
  assumes prefix_of π σ and prefix_of π σ' and i < progress σ φ (plen π)
  shows MFOTL.sat σ v i φ ↔ MFOTL.sat σ' v i φ
  using assms(3) proof (induction φ arbitrary: v i)
  case (Pred e ts)
  with Γ_prefix_conv[OF assms(1,2)] show ?case by simp
  next
  case (Eq t1 t2)
  show ?case by simp
  next
  case (Neg φ)
  then show ?case by simp
  next
  case (Or φ1 φ2)
  then show ?case by simp

```

```

next
  case (Exists  $\varphi$ )
  then show ?case by simp
next
  case (Prev I  $\varphi$ )
  with  $\tau\_prefix\_conv[OF\ assms(1,2)]$  show ?case
  by (cases i) (auto split: if_splits)
next
  case (Next I  $\varphi$ )
  then have  $Suc\ i < plen\ \pi$ 
  by (auto intro: order.strict_trans2[OF __progress_le[of  $\sigma\ \varphi$ ]])
  with Next  $\tau\_prefix\_conv[OF\ assms(1,2)]$  show ?case by simp
next
  case (Since  $\varphi1\ I\ \varphi2$ )
  then have  $i < plen\ \pi$ 
  by (auto elim!: order.strict_trans2[OF __progress_le])
  with Since  $\tau\_prefix\_conv[OF\ assms(1,2)]$  show ?case by auto
next
  case (Until  $\varphi1\ I\ \varphi2$ )
  from Until.prem1 obtain b where [simp]:  $right\ I = enat\ b$ 
  by (cases right I) (auto simp add: Inf_UNIV_nat)
  from Until.prem2 obtain j where  $\tau\ \sigma\ i + b + 1 \leq \tau\ \sigma\ j$ 
   $j \leq progress\ \sigma\ \varphi1\ (plen\ \pi)$   $j \leq progress\ \sigma\ \varphi2\ (plen\ \pi)$ 
  by atomize_elim (auto 0 4 simp add: less_eq_Suc_le not_le intro: Suc_leI dest: spec[of __ i]
  dest!: le_cInf_iff[THEN iffD1, rotated -1])
  then have 1:  $k < progress\ \sigma\ \varphi1\ (plen\ \pi)$  and 2:  $k < progress\ \sigma\ \varphi2\ (plen\ \pi)$ 
  if  $\tau\ \sigma\ k \leq \tau\ \sigma\ i + b$  for k
  using that by (fastforce elim!: order.strict_trans2[rotated] intro: less_τD[of  $\sigma$ ])+
  have 3:  $k < plen\ \pi$  if  $\tau\ \sigma\ k \leq \tau\ \sigma\ i + b$  for k
  using 1[OF that] by (simp add: less_eq_Suc_le order.trans[OF __progress_le])

  from Until.prem3 have  $i < progress\ \sigma'\ (MFOTL.Until\ \varphi1\ I\ \varphi2)\ (plen\ \pi)$ 
  unfolding progress_prefix_conv[OF assms(1,2)] .
  then obtain j where  $\tau\ \sigma'\ i + b + 1 \leq \tau\ \sigma'\ j$ 
   $j \leq progress\ \sigma'\ \varphi1\ (plen\ \pi)$   $j \leq progress\ \sigma'\ \varphi2\ (plen\ \pi)$ 
  by atomize_elim (auto 0 4 simp add: less_eq_Suc_le not_le intro: Suc_leI dest: spec[of __ i]
  dest!: le_cInf_iff[THEN iffD1, rotated -1])
  then have 11:  $k < progress\ \sigma\ \varphi1\ (plen\ \pi)$  and 21:  $k < progress\ \sigma\ \varphi2\ (plen\ \pi)$ 
  if  $\tau\ \sigma'\ k \leq \tau\ \sigma'\ i + b$  for k
  unfolding progress_prefix_conv[OF assms(1,2)]
  using that by (fastforce elim!: order.strict_trans2[rotated] intro: less_τD[of  $\sigma'$ ])+
  have 31:  $k < plen\ \pi$  if  $\tau\ \sigma'\ k \leq \tau\ \sigma'\ i + b$  for k
  using 11[OF that] by (simp add: less_eq_Suc_le order.trans[OF __progress_le])

  show ?case unfolding sat_simps
  proof ((intro ex_cong iffI; elim conjE), goal_cases LR RL)
  case (LR j)
  with Until.IH(1)[OF 1] Until.IH(2)[OF 2]  $\tau\_prefix\_conv[OF\ assms(1,2)\ 3]$  show ?case
  by (auto 0 4 simp: le_diff_conv add.commute dest: less_imp_le order.trans[OF  $\tau\_mono$ , rotated])
  next
  case (RL j)
  with Until.IH(1)[OF 11] Until.IH(2)[OF 21]  $\tau\_prefix\_conv[OF\ assms(1,2)\ 31]$  show ?case
  by (auto 0 4 simp: le_diff_conv add.commute dest: less_imp_le order.trans[OF  $\tau\_mono$ , rotated])
  qed
qed

```

## 6.4 Specification

**definition**  $pprogress :: 'a \text{ MFOTL.formula} \Rightarrow 'a \text{ MFOTL.prefix} \Rightarrow \text{nat}$  **where**  
 $pprogress \varphi \pi = (\text{THE } n. \forall \sigma. \text{prefix\_of } \pi \sigma \longrightarrow \text{progress } \sigma \varphi (\text{plen } \pi) = n)$

**lemma**  $pprogress\_eq: \text{prefix\_of } \pi \sigma \Longrightarrow pprogress \varphi \pi = \text{progress } \sigma \varphi (\text{plen } \pi)$   
**unfolding**  $pprogress\_def$  **using**  $\text{progress\_prefix\_conv}$   
**by**  $\text{blast}$

**locale**  $\text{future\_bounded\_mfotl} =$   
**fixes**  $\varphi :: 'a \text{ MFOTL.formula}$   
**assumes**  $\text{future\_bounded}: \text{MFOTL.future\_reach } \varphi \neq \infty$

**sublocale**  $\text{future\_bounded\_mfotl} \subseteq \text{sliceable\_timed\_progress } \text{MFOTL.nfv } \varphi \text{ MFOTL.fv } \varphi \text{ relevant\_events}$

$\varphi$   
 $\lambda \sigma v i. \text{MFOTL.sat } \sigma v i \varphi pprogress \varphi$   
**proof** ( $\text{unfold\_locales}, \text{goal\_cases}$ )  
**case** (1  $x$ )  
**then show**  $?case$  **by** ( $\text{simp add: fvi\_less\_nfv}$ )  
**next**  
**case** (2  $v v' \sigma i$ )  
**then show**  $?case$  **by** ( $\text{simp cong: sat\_fvi\_cong}[\text{rule\_format}]$ )  
**next**  
**case** (3  $v S \sigma i$ )  
**then show**  $?case$  **using**  $\text{sat\_slice\_iff}[\text{of } v, \text{symmetric}]$  **by**  $\text{simp}$   
**next**  
**case** (4  $\pi \pi'$ )  
**moreover obtain**  $\sigma$  **where**  $\text{prefix\_of } \pi' \sigma$   
**using**  $\text{ex\_prefix\_of } ..$   
**moreover have**  $\text{prefix\_of } \pi \sigma$   
**using**  $\text{prefix\_of\_antimono}[\text{OF } \langle \pi \leq \pi' \rangle \langle \text{prefix\_of } \pi' \sigma \rangle]$ .  
**ultimately show**  $?case$   
**by** ( $\text{simp add: pprogress\_eq plen\_mono progress\_mono}$ )  
**next**  
**case** (5  $\sigma x$ )  
**obtain**  $j$  **where**  $x \leq \text{progress } \sigma \varphi j$   
**using**  $\text{future\_bounded progress\_ge}$  **by**  $\text{blast}$   
**then have**  $x \leq pprogress \varphi (\text{take\_prefix } j \sigma)$   
**by** ( $\text{simp add: pprogress\_eq}[\text{of } \_ \sigma]$ )  
**then show**  $?case$  **by**  $\text{force}$   
**next**  
**case** (6  $\pi \sigma \sigma' i v$ )  
**then have**  $i < \text{progress } \sigma \varphi (\text{plen } \pi)$   
**by** ( $\text{simp add: pprogress\_eq}$ )  
**with** 6 **show**  $?case$   
**using**  $\text{sat\_prefix\_conv}$  **by**  $\text{blast}$   
**next**  
**case** (7  $\pi \pi'$ )  
**then have**  $\text{plen } \pi = \text{plen } \pi'$   
**by**  $\text{transfer } (\text{simp add: list\_eq\_iff\_nth\_eq})$   
**moreover obtain**  $\sigma \sigma'$  **where**  $\text{prefix\_of } \pi \sigma \text{ prefix\_of } \pi' \sigma'$   
**using**  $\text{ex\_prefix\_of}$  **by**  $\text{blast+}$   
**moreover have**  $\forall i < \text{plen } \pi. \tau \sigma i = \tau \sigma' i$   
**using** 7  $\text{calculation}$   
**by**  $\text{transfer } (\text{simp add: list\_eq\_iff\_nth\_eq})$   
**ultimately show**  $?case$   
**by** ( $\text{simp add: pprogress\_eq progress\_time\_conv}$ )  
**qed**

**locale** *monitorable\_mfotl* =  
**fixes**  $\varphi :: 'a \text{ MFOTL.formula}$   
**assumes** *monitorable*: *mmonitorable*  $\varphi$

**sublocale** *monitorable\_mfotl*  $\subseteq$  *future\_bounded\_mfotl*  
**using** *monitorable* **by** *unfold\_locales* (*simp add: mmonitorable\_def*)

## 6.5 Correctness

### 6.5.1 Invariants

**definition** *wf\_mbuf2* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \Rightarrow 'a \text{ table} \Rightarrow \text{bool}) \Rightarrow (\text{nat} \Rightarrow 'a \text{ table} \Rightarrow \text{bool}) \Rightarrow 'a \text{ mbuf2} \Rightarrow \text{bool}$  **where**  
*wf\_mbuf2*  $i \ j \ P \ Q \ \text{buf} \longleftrightarrow i \leq j \wedge i \leq j \wedge (\text{case } \text{buf} \text{ of } (xs, ys) \Rightarrow \text{list\_all2 } P \ [i..<j] \ xs \wedge \text{list\_all2 } Q \ [i..<j] \ ys)$

**definition** *wf\_mbuf2'* ::  $'a \text{ MFOTL.trace} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ list set} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow 'a \text{ mbuf2} \Rightarrow \text{bool}$  **where**  
*wf\_mbuf2'*  $\sigma \ j \ n \ R \ \varphi \ \psi \ \text{buf} \longleftrightarrow \text{wf\_mbuf2} \ (\min \ (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)) \ (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)$   
 $(\lambda i. \text{qtable } n \ (\text{MFOTL.fv } \varphi) \ (\text{mem\_restr } R) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \varphi))$   
 $(\lambda i. \text{qtable } n \ (\text{MFOTL.fv } \psi) \ (\text{mem\_restr } R) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \psi)) \ \text{buf}$

**lemma** *wf\_mbuf2'\_UNIV\_alt*: *wf\_mbuf2'*  $\sigma \ j \ n \ \text{UNIV } \varphi \ \psi \ \text{buf} \longleftrightarrow (\text{case } \text{buf} \text{ of } (xs, ys) \Rightarrow \text{list\_all2} \ (\lambda i. \text{wf\_table } n \ (\text{MFOTL.fv } \varphi) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \varphi)) \ [\min \ (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)] \ ..< \ (\text{progress } \sigma \ \varphi \ j)] \ xs \wedge \text{list\_all2} \ (\lambda i. \text{wf\_table } n \ (\text{MFOTL.fv } \psi) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \psi)) \ [\min \ (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)] \ ..< \ (\text{progress } \sigma \ \psi \ j)] \ ys)$

**unfolding** *wf\_mbuf2'\_def* *wf\_mbuf2\_def*

**by** (*simp add: mem\_restr\_UNIV[THEN eqTrueI, abs\_def]* *split: prod.split*)

**definition** *wf\_ts* ::  $'a \text{ MFOTL.trace} \Rightarrow \text{nat} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow \text{ts list} \Rightarrow \text{bool}$  **where**

*wf\_ts*  $\sigma \ j \ \varphi \ \psi \ ts \longleftrightarrow \text{list\_all2} \ (\lambda i \ t. t = \tau \sigma \ i) \ [\min \ (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)] \ ..< [j] \ ts$

**abbreviation** *Since\_pos*  $\varphi \ I \ \psi \equiv \text{MFOTL.Since} \ (\text{if } \text{pos} \text{ then } \varphi \ \text{else } \text{MFOTL.Neg } \varphi) \ I \ \psi$

**definition** *wf\_since\_aux* ::  $'a \text{ MFOTL.trace} \Rightarrow \text{nat} \Rightarrow 'a \text{ list set} \Rightarrow \text{bool} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow \mathcal{I} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow 'a \text{ msaux} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**  
*wf\_since\_aux*  $\sigma \ n \ R \ \text{pos } \varphi \ I \ \psi \ \text{aux} \ ne \longleftrightarrow \text{sorted\_wrt} \ (\lambda x \ y. \text{fst } x > \text{fst } y) \ \text{aux} \wedge (\forall t \ X. (t, X) \in \text{set } \text{aux} \longrightarrow ne \neq 0 \wedge t \leq \tau \sigma \ (ne-1) \wedge \tau \sigma \ (ne-1) - t \leq \text{right } I \wedge (\exists i. \tau \sigma \ i = t)) \wedge \text{qtable } n \ (\text{MFOTL.fv } \psi) \ (\text{mem\_restr } R) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ (ne-1) \ (\text{Since\_pos } \varphi \ (\text{point } (\tau \sigma \ (ne-1) - t)) \ \psi)) \ X) \wedge (\forall t. ne \neq 0 \wedge t \leq \tau \sigma \ (ne-1) \wedge \tau \sigma \ (ne-1) - t \leq \text{right } I \wedge (\exists i. \tau \sigma \ i = t) \longrightarrow (\exists X. (t, X) \in \text{set } \text{aux}))$

**lemma** *qtable\_mem\_restr\_UNIV*: *qtable*  $n \ A \ (\text{mem\_restr } \text{UNIV}) \ Q \ X = \text{wf\_table } n \ A \ Q \ X$   
**unfolding** *qtable\_def* **by** *auto*

**lemma** *wf\_since\_aux\_UNIV\_alt*:

*wf\_since\_aux*  $\sigma \ n \ \text{UNIV } \text{pos } \varphi \ I \ \psi \ \text{aux} \ ne \longleftrightarrow \text{sorted\_wrt} \ (\lambda x \ y. \text{fst } x > \text{fst } y) \ \text{aux} \wedge (\forall t \ X. (t, X) \in \text{set } \text{aux} \longrightarrow ne \neq 0 \wedge t \leq \tau \sigma \ (ne-1) \wedge \tau \sigma \ (ne-1) - t \leq \text{right } I \wedge (\exists i. \tau \sigma \ i = t)) \wedge \text{wf\_table } n \ (\text{MFOTL.fv } \psi) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ (ne-1) \ (\text{Since\_pos } \varphi \ (\text{point } (\tau \sigma \ (ne-1) - t)) \ \psi)) \ X) \wedge (\forall t. ne \neq 0 \wedge t \leq \tau \sigma \ (ne-1) \wedge \tau \sigma \ (ne-1) - t \leq \text{right } I \wedge (\exists i. \tau \sigma \ i = t) \longrightarrow (\exists X. (t, X) \in \text{set } \text{aux}))$

**unfolding** *wf\_since\_aux\_def* *qtable\_mem\_restr\_UNIV* ..

**definition**  $wf\_until\_aux :: 'a MFOTL.trace \Rightarrow nat \Rightarrow 'a list set \Rightarrow bool \Rightarrow 'a MFOTL.formula \Rightarrow \mathcal{I} \Rightarrow 'a MFOTL.formula \Rightarrow 'a muaux \Rightarrow nat \Rightarrow bool$  **where**  
 $wf\_until\_aux \sigma n R pos \varphi I \psi aux ne \longleftrightarrow list\_all2 (\lambda x i. case x of (t, r1, r2) \Rightarrow t = \tau \sigma i \wedge$   
 $qtable n (MFOTL.fv \varphi) (mem\_restr R) (\lambda v. if pos then (\forall k \in \{i..<ne+length\ aux\}. MFOTL.sat \sigma$   
 $(map\ the\ v) k \varphi)$   
 $else (\exists k \in \{i..<ne+length\ aux\}. MFOTL.sat \sigma (map\ the\ v) k \varphi)) r1 \wedge$   
 $qtable n (MFOTL.fv \psi) (mem\_restr R) (\lambda v. (\exists j. i \leq j \wedge j < ne + length\ aux \wedge mem (\tau \sigma j - \tau \sigma$   
 $i) I \wedge$   
 $MFOTL.sat \sigma (map\ the\ v) j \psi \wedge$   
 $(\forall k \in \{i..<j\}. if pos then MFOTL.sat \sigma (map\ the\ v) k \varphi else \neg MFOTL.sat \sigma (map\ the\ v) k \varphi)))$   
 $r2)$   
 $aux [ne..<ne+length\ aux]$

**lemma**  $wf\_until\_aux\_UNIV\_alt:$

$wf\_until\_aux \sigma n UNIV pos \varphi I \psi aux ne \longleftrightarrow list\_all2 (\lambda x i. case x of (t, r1, r2) \Rightarrow t = \tau \sigma i \wedge$   
 $wf\_table n (MFOTL.fv \varphi) (\lambda v. if pos$   
 $then (\forall k \in \{i..<ne+length\ aux\}. MFOTL.sat \sigma (map\ the\ v) k \varphi)$   
 $else (\exists k \in \{i..<ne+length\ aux\}. MFOTL.sat \sigma (map\ the\ v) k \varphi)) r1 \wedge$   
 $wf\_table n (MFOTL.fv \psi) (\lambda v. \exists j. i \leq j \wedge j < ne + length\ aux \wedge mem (\tau \sigma j - \tau \sigma i) I \wedge$   
 $MFOTL.sat \sigma (map\ the\ v) j \psi \wedge$   
 $(\forall k \in \{i..<j\}. if pos then MFOTL.sat \sigma (map\ the\ v) k \varphi else \neg MFOTL.sat \sigma (map\ the\ v) k \varphi)))$   
 $r2)$   
 $aux [ne..<ne+length\ aux]$   
**unfolding**  $wf\_until\_aux\_def\ qtable\_mem\_restr\_UNIV ..$

**inductive**  $wf\_mformula :: 'a MFOTL.trace \Rightarrow nat \Rightarrow$

$nat \Rightarrow 'a list set \Rightarrow 'a mformula \Rightarrow 'a MFOTL.formula \Rightarrow bool$   
**for**  $\sigma j$  **where**  
 $Eq: MFOTL.is\_Const t1 \vee MFOTL.is\_Const t2 \Longrightarrow$   
 $\forall x \in MFOTL.fv\_trm t1. x < n \Longrightarrow \forall x \in MFOTL.fv\_trm t2. x < n \Longrightarrow$   
 $wf\_mformula \sigma j n R (MRel (eq\_rel n t1 t2)) (MFOTL.Eq t1 t2)$   
 $| neq\_Const: \varphi = MRel (neq\_rel n (MFOTL.Const x) (MFOTL.Const y)) \Longrightarrow$   
 $wf\_mformula \sigma j n R \varphi (MFOTL.Neg (MFOTL.Eq (MFOTL.Const x) (MFOTL.Const y)))$   
 $| neq\_Var: x < n \Longrightarrow$   
 $wf\_mformula \sigma j n R (MRel empty\_table) (MFOTL.Neg (MFOTL.Eq (MFOTL.Var x) (MFOTL.Var$   
 $x)))$   
 $| Pred: \forall x \in MFOTL.fv (MFOTL.Pred e ts). x < n \Longrightarrow$   
 $wf\_mformula \sigma j n R (MPred e ts) (MFOTL.Pred e ts)$   
 $| And: wf\_mformula \sigma j n R \varphi \varphi' \Longrightarrow wf\_mformula \sigma j n R \psi \psi' \Longrightarrow$   
 $if pos then \chi = MFOTL.And \varphi' \psi' \wedge \neg (safe\_formula (MFOTL.Neg \psi') \wedge MFOTL.fv \psi' \subseteq MFOTL.fv$   
 $\varphi')$   
 $else \chi = MFOTL.And\_Not \varphi' \psi' \wedge safe\_formula \psi' \wedge MFOTL.fv \psi' \subseteq MFOTL.fv \varphi' \Longrightarrow$   
 $wf\_mbuf2' \sigma j n R \varphi' \psi' buf \Longrightarrow$   
 $wf\_mformula \sigma j n R (MAnd \varphi pos \psi buf) \chi$   
 $| Or: wf\_mformula \sigma j n R \varphi \varphi' \Longrightarrow wf\_mformula \sigma j n R \psi \psi' \Longrightarrow$   
 $MFOTL.fv \varphi' = MFOTL.fv \psi' \Longrightarrow$   
 $wf\_mbuf2' \sigma j n R \varphi' \psi' buf \Longrightarrow$   
 $wf\_mformula \sigma j n R (MOr \varphi \psi buf) (MFOTL.Or \varphi' \psi')$   
 $| Exists: wf\_mformula \sigma j (Suc n) (lift\_envs R) \varphi \varphi' \Longrightarrow$   
 $wf\_mformula \sigma j n R (MExists \varphi) (MFOTL.Exists \varphi')$   
 $| Prev: wf\_mformula \sigma j n R \varphi \varphi' \Longrightarrow$   
 $first \longleftrightarrow j = 0 \Longrightarrow$   
 $list\_all2 (\lambda i. qtable n (MFOTL.fv \varphi') (mem\_restr R) (\lambda v. MFOTL.sat \sigma (map\ the\ v) i \varphi'))$   
 $[min (progress \sigma \varphi' j) (j-1)..<progress \sigma \varphi' j] buf \Longrightarrow$   
 $list\_all2 (\lambda i t. t = \tau \sigma i) [min (progress \sigma \varphi' j) (j-1)..<j] nts \Longrightarrow$   
 $wf\_mformula \sigma j n R (MPrev I \varphi first buf nts) (MFOTL.Prev I \varphi')$

| *Next*:  $wf\_mformula \sigma j n R \varphi \varphi' \implies$   
 $first \longleftrightarrow progress \sigma \varphi' j = 0 \implies$   
 $list\_all2 (\lambda i t. t = \tau \sigma i) [progress \sigma \varphi' j - 1..<j] nts \implies$   
 $wf\_mformula \sigma j n R (MNext I \varphi first nts) (MFOTL.Next I \varphi')$   
| *Since*:  $wf\_mformula \sigma j n R \varphi \varphi' \implies wf\_mformula \sigma j n R \psi \psi' \implies$   
 $if\ pos\ then\ \varphi'' = \varphi' \ else\ \varphi'' = MFOTL.Neg\ \varphi' \implies$   
 $safe\_formula \varphi'' = pos \implies$   
 $MFOTL.fv \varphi' \subseteq MFOTL.fv \psi' \implies$   
 $wf\_mbuf2' \sigma j n R \varphi' \psi' buf \implies$   
 $wf\_ts \sigma j \varphi' \psi' nts \implies$   
 $wf\_since\_aux \sigma n R pos \varphi' I \psi' aux (progress \sigma (MFOTL.Since \varphi'' I \psi') j) \implies$   
 $wf\_mformula \sigma j n R (MSince pos \varphi I \psi buf nts aux) (MFOTL.Since \varphi'' I \psi')$   
| *Until*:  $wf\_mformula \sigma j n R \varphi \varphi' \implies wf\_mformula \sigma j n R \psi \psi' \implies$   
 $if\ pos\ then\ \varphi'' = \varphi' \ else\ \varphi'' = MFOTL.Neg\ \varphi' \implies$   
 $safe\_formula \varphi'' = pos \implies$   
 $MFOTL.fv \varphi' \subseteq MFOTL.fv \psi' \implies$   
 $wf\_mbuf2' \sigma j n R \varphi' \psi' buf \implies$   
 $wf\_ts \sigma j \varphi' \psi' nts \implies$   
 $wf\_until\_aux \sigma n R pos \varphi' I \psi' aux (progress \sigma (MFOTL.Until \varphi'' I \psi') j) \implies$   
 $progress \sigma (MFOTL.Until \varphi'' I \psi') j + length\ aux = \min (progress \sigma \varphi' j) (progress \sigma \psi' j) \implies$   
 $wf\_mformula \sigma j n R (MUntil pos \varphi I \psi buf nts aux) (MFOTL.Until \varphi'' I \psi')$

**definition**  $wf\_mstate :: 'a MFOTL.formula \Rightarrow 'a MFOTL.prefix \Rightarrow 'a list\ set \Rightarrow 'a\ mstate \Rightarrow bool$  **where**  
 $wf\_mstate \varphi \pi R st \longleftrightarrow mstate\_n\ st = MFOTL.nfv \varphi \wedge (\forall \sigma. prefix\_of \pi \sigma \longrightarrow$   
 $mstate\_i\ st = progress \sigma \varphi (plen \pi) \wedge$   
 $wf\_mformula \sigma (plen \pi) (mstate\_n\ st) R (mstate\_m\ st) \varphi)$

## 6.5.2 Initialisation

**lemma**  $minit0\_And: \neg (safe\_formula (MFOTL.Neg \psi) \wedge MFOTL.fv \psi \subseteq MFOTL.fv \varphi) \implies$   
 $minit0\ n (MFOTL.And \varphi \psi) = MAnd (minit0\ n \varphi) True (minit0\ n \psi) ([], [])$   
**unfolding**  $MFOTL.And\_def$  **by** *simp*

**lemma**  $minit0\_And\_Not: safe\_formula \psi \wedge MFOTL.fv \psi \subseteq MFOTL.fv \varphi \implies$   
 $minit0\ n (MFOTL.And\_Not \varphi \psi) = (MAnd (minit0\ n \varphi) False (minit0\ n \psi) ([], []))$   
**unfolding**  $MFOTL.And\_Not\_def$   $MFOTL.is\_Neg\_def$  **by** (*simp split: formula.split*)

**lemma**  $wf\_mbuf2'\_0: wf\_mbuf2' \sigma 0 n R \varphi \psi ([], [])$   
**unfolding**  $wf\_mbuf2'\_def$   $wf\_mbuf2\_def$  **by** *simp*

**lemma**  $wf\_ts\_0: wf\_ts \sigma 0 \varphi \psi []$   
**unfolding**  $wf\_ts\_def$  **by** *simp*

**lemma**  $wf\_since\_aux\_Nil: wf\_since\_aux \sigma n R pos \varphi' I \psi' [] 0$   
**unfolding**  $wf\_since\_aux\_def$  **by** *simp*

**lemma**  $wf\_until\_aux\_Nil: wf\_until\_aux \sigma n R pos \varphi' I \psi' [] 0$   
**unfolding**  $wf\_until\_aux\_def$  **by** *simp*

**lemma**  $wf\_minit0: safe\_formula \varphi \implies \forall x \in MFOTL.fv \varphi. x < n \implies$   
 $wf\_mformula \sigma 0 n R (minit0\ n \varphi) \varphi$   
**by** (*induction arbitrary: n R rule: safe\\_formula\\_induct*)  
*(auto simp add: minit0\\_And fvi\\_And minit0\\_And\\_Not fvi\\_And\\_Not*  
*intro!: wf\\_mformula.intros wf\\_mbuf2'\\_0 wf\\_ts\\_0 wf\\_since\\_aux\\_Nil wf\\_until\\_aux\\_Nil*  
*dest: fvi\\_Suc\\_bound)*

**lemma**  $wf\_mstate\_minit: safe\_formula \varphi \implies wf\_mstate \varphi pnil R (minit \varphi)$   
**unfolding**  $wf\_mstate\_def$   $minit\_def$   $Let\_def$   
**by** (*auto intro!: wf\\_minit0 fvi\\_less\\_nfv*)

### 6.5.3 Evaluation

**lemma** *match\_wf\_tuple*: *Some f = match ts xs  $\implies$  wf\_tuple n ( $\bigcup t \in \text{set } ts. \text{MFOTL.fv\_trm } t$ ) (tabulate f 0 n)*

**by** (*induction ts xs arbitrary; f rule: match.induct*)  
*(fastforce simp: wf\_tuple\_def split: if\_splits option.splits)+*

**lemma** *match\_fvi\_trm\_None*: *Some f = match ts xs  $\implies \forall t \in \text{set } ts. x \notin \text{MFOTL.fv\_trm } t \implies f x = \text{None}$*

**by** (*induction ts xs arbitrary; f rule: match.induct*) (*auto split: if\_splits option.splits*)

**lemma** *match\_fvi\_trm\_Some*: *Some f = match ts xs  $\implies t \in \text{set } ts \implies x \in \text{MFOTL.fv\_trm } t \implies f x \neq \text{None}$*

**by** (*induction ts xs arbitrary; f rule: match.induct*) (*auto split: if\_splits option.splits*)

**lemma** *match\_eval\_trm*:  $\forall t \in \text{set } ts. \forall i \in \text{MFOTL.fv\_trm } t. i < n \implies \text{Some } f = \text{match } ts \text{ xs} \implies \text{map } (\text{MFOTL.eval\_trm } (\text{tabulate } (\lambda i. \text{the } (f \ i)) \ 0 \ n)) \ ts = \text{xs}$

**proof** (*induction ts xs arbitrary; f rule: match.induct*)

**case** ( $\exists x \ ts \ y \ ys$ )

**from**  $\exists(1)[\text{symmetric}] \exists(2,3)$  **show** *?case*

**by** (*auto 0 3 dest: match\_fvi\_trm\_Some sym split: option.splits if\_splits intro!: eval\_trm\_cong*)

**qed** (*auto split: if\_splits*)

**lemma** *wf\_tuple\_tabulate\_Some*: *wf\_tuple n A (tabulate f 0 n)  $\implies x \in A \implies x < n \implies \exists y. f x = \text{Some } y$*

**unfolding** *wf\_tuple\_def* **by** *auto*

**lemma** *ex\_match*: *wf\_tuple n ( $\bigcup t \in \text{set } ts. \text{MFOTL.fv\_trm } t$ ) v  $\implies \forall t \in \text{set } ts. \forall x \in \text{MFOTL.fv\_trm } t. x < n \implies$*

$\exists f. \text{match } ts \ (\text{map } (\text{MFOTL.eval\_trm } (\text{map the } v)) \ ts) = \text{Some } f \wedge v = \text{tabulate } f \ 0 \ n$

**proof** (*induction ts map (MFOTL.eval\_trm (map the v)) ts arbitrary; v rule: match.induct*)

**case** ( $\exists x \ ts \ y \ ys$ )

**then show** *?case*

**proof** (*cases x  $\in$  ( $\bigcup t \in \text{set } ts. \text{MFOTL.fv\_trm } t$ )*)

**case** *True*

**with**  $\exists$  **show** *?thesis*

**by** (*auto simp: insert\_absorb dest!: wf\_tuple\_tabulate\_Some meta\_spec[of \_ v]*)

**next**

**case** *False*

**with**  $\exists(3,4)$  **have**

*\*: map (MFOTL.eval\_trm (map the v)) ts = map (MFOTL.eval\_trm (map the (v[x := None]))) ts*

**by** (*auto simp: wf\_tuple\_def nth\_list\_update intro!: eval\_trm\_cong*)

**from** *False*  $\exists(2-4)$  **obtain** *f where*

*match ts (map (MFOTL.eval\_trm (map the v)) ts) = Some f v[x := None] = tabulate f 0 n*

**unfolding** *\**

**by** (*atomize\_elim, intro*  $\exists(1)[\text{of } v[x := \text{None}]]$ )

(*auto simp: wf\_tuple\_def nth\_list\_update intro!: eval\_trm\_cong*)

**moreover from** *False this have* *f x = None length v = n*

**by** (*auto dest: match\_fvi\_trm\_None[OF sym] arg\_cong[of \_ \_ length]*)

**ultimately show** *?thesis using*  $\exists(3)$

**by** (*auto simp: list\_eq\_iff\_nth\_eq wf\_tuple\_def*)

**qed**

**qed** (*auto simp: wf\_tuple\_def intro: nth\_equalityI*)

**lemma** *eq\_rel\_eval\_trm*: *v  $\in$  eq\_rel n t1 t2  $\implies \text{MFOTL.is\_Const } t1 \vee \text{MFOTL.is\_Const } t2 \implies \forall x \in \text{MFOTL.fv\_trm } t1 \cup \text{MFOTL.fv\_trm } t2. x < n \implies$*

*MFOTL.eval\_trm (map the v) t1 = MFOTL.eval\_trm (map the v) t2*

**by** (*cases t1; cases t2*) (*simp\_all add: singleton\_table\_def split: if\_splits*)

**lemma** *in\_eq\_rel*:  $wf\_tuple\ n\ (MFOTL.fv\_trm\ t1\ \cup\ MFOTL.fv\_trm\ t2)\ v\ \Longrightarrow$   
 $MFOTL.is\_Const\ t1\ \vee\ MFOTL.is\_Const\ t2\ \Longrightarrow$   
 $MFOTL.eval\_trm\ (map\ the\ v)\ t1\ =\ MFOTL.eval\_trm\ (map\ the\ v)\ t2\ \Longrightarrow$   
 $v\ \in\ eq\_rel\ n\ t1\ t2$   
**by** (*cases* *t1*; *cases* *t2*)  
*(auto simp: singleton\_table\_def wf\_tuple\_def unit\_table\_def intro!: nth\_equalityI split: if\_splits)*

**lemma** *table\_eq\_rel*:  $MFOTL.is\_Const\ t1\ \vee\ MFOTL.is\_Const\ t2\ \Longrightarrow$   
 $table\ n\ (MFOTL.fv\_trm\ t1\ \cup\ MFOTL.fv\_trm\ t2)\ (eq\_rel\ n\ t1\ t2)$   
**by** (*cases* *t1*; *cases* *t2*; *simp*)

**lemma** *wf\_tuple\_Suc\_fviD*:  $wf\_tuple\ (Suc\ n)\ (MFOTL.fvi\ b\ \varphi)\ v\ \Longrightarrow\ wf\_tuple\ n\ (MFOTL.fvi\ (Suc\ b)\ \varphi)\ (tl\ v)$   
**unfolding** *wf\_tuple\_def* **by** (*simp* *add: fvi\_Suc\_nth\_tl*)

**lemma** *table\_fvi\_tl*:  $table\ (Suc\ n)\ (MFOTL.fvi\ b\ \varphi)\ X\ \Longrightarrow\ table\ n\ (MFOTL.fvi\ (Suc\ b)\ \varphi)\ (tl\ 'X)$   
**unfolding** *table\_def* **by** (*auto* *intro: wf\_tuple\_Suc\_fviD*)

**lemma** *wf\_tuple\_Suc\_fvi\_SomeI*:  $0\ \in\ MFOTL.fvi\ b\ \varphi\ \Longrightarrow\ wf\_tuple\ n\ (MFOTL.fvi\ (Suc\ b)\ \varphi)\ v\ \Longrightarrow$   
 $wf\_tuple\ (Suc\ n)\ (MFOTL.fvi\ b\ \varphi)\ (Some\ x\ \# v)$   
**unfolding** *wf\_tuple\_def*  
**by** (*auto* *simp: fvi\_Suc\_less\_Suc\_eq\_0\_disj*)

**lemma** *wf\_tuple\_Suc\_fvi\_NoneI*:  $0\ \notin\ MFOTL.fvi\ b\ \varphi\ \Longrightarrow\ wf\_tuple\ n\ (MFOTL.fvi\ (Suc\ b)\ \varphi)\ v\ \Longrightarrow$   
 $wf\_tuple\ (Suc\ n)\ (MFOTL.fvi\ b\ \varphi)\ (None\ \# v)$   
**unfolding** *wf\_tuple\_def*  
**by** (*auto* *simp: fvi\_Suc\_less\_Suc\_eq\_0\_disj*)

**lemma** *qtable\_project\_fv*:  $qtable\ (Suc\ n)\ (fv\ \varphi)\ (mem\_restr\ (lift\_envs\ R))\ P\ X\ \Longrightarrow$   
 $qtable\ n\ (MFOTL.fvi\ (Suc\ 0)\ \varphi)\ (mem\_restr\ R)$   
*( $\lambda v. \exists x. P\ ((if\ 0\ \in\ fv\ \varphi\ then\ Some\ x\ else\ None)\ \# v))\ (tl\ 'X)$ )  
**using** *neq0\_conv* **by** (*fastforce* *simp: image\_iff Bex\_def fvi\_Suc\_elim!: qtable\_cong dest!: qtable\_project*)*

**lemma** *mprev*:  $mprev\_next\ I\ xs\ ts\ =\ (ys,\ xs',\ ts')\ \Longrightarrow$   
 $list\_all2\ P\ [i..<j]\ xs\ \Longrightarrow\ list\_all2\ (\lambda i\ t. t = \tau\ \sigma\ i)\ [i..<j]\ ts\ \Longrightarrow\ i \leq j' \Longrightarrow i < j \Longrightarrow$   
 $list\_all2\ (\lambda i\ X. if\ mem\ (\tau\ \sigma\ (Suc\ i))\ -\ \tau\ \sigma\ i)\ I\ then\ P\ i\ X\ else\ X = empty\_table)$   
 $[i..<min\ j'\ (j-1)]\ ys\ \wedge$   
 $list\_all2\ P\ [min\ j'\ (j-1)..<j]\ xs'\ \wedge$   
 $list\_all2\ (\lambda i\ t. t = \tau\ \sigma\ i)\ [min\ j'\ (j-1)..<j]\ ts'$   
**proof** (*induction* *I xs ts arbitrary: i ys xs' ts' rule: mprev\_next.induct*)  
**case** (*1 I ts*)  
**then** *have*  $min\ j'\ (j-1) = i$  **by** *auto*  
**with** *1* **show** *?case* **by** *auto*  
**next**  
**case** (*3 I v v' t*)  
**then** *have*  $min\ j'\ (j-1) = i$  **by** (*auto* *simp: list\_all2\_Cons2 upt\_eq\_Cons\_conv*)  
**with** *3* **show** *?case* **by** *auto*  
**next**  
**case** (*4 I x xs t t' ts*)  
**from** *4(1)[of tl ys xs' ts' Suc i]* *4(2-6)* **show** *?case*  
**by** (*auto* *simp* *add: list\_all2\_Cons2 upt\_eq\_Cons\_conv Suc\_less\_eq2*  
*elim!: list\_rel\_mono\_strong split: prod\_splits if\_splits*)  
**qed** *simp*

**lemma** *mnext*:  $mprev\_next\ I\ xs\ ts\ =\ (ys,\ xs',\ ts')\ \Longrightarrow$   
 $list\_all2\ P\ [Suc\ i..<j]\ xs\ \Longrightarrow\ list\_all2\ (\lambda i\ t. t = \tau\ \sigma\ i)\ [i..<j]\ ts\ \Longrightarrow\ Suc\ i \leq j' \Longrightarrow i < j \Longrightarrow$   
 $list\_all2\ (\lambda i\ X. if\ mem\ (\tau\ \sigma\ (Suc\ i))\ -\ \tau\ \sigma\ i)\ I\ then\ P\ (Suc\ i)\ X\ else\ X = empty\_table)$   
 $[i..<min\ (j'-1)\ (j-1)]\ ys\ \wedge$

```

list_all2 P [Suc (min (j'-1) (j-1))..<j'] xs' ∧
list_all2 (λi t. t = τ σ i) [min (j'-1) (j-1)..<j] ts'
proof (induction I xs ts arbitrary: i ys xs' ts' rule: mprev_next.induct)
  case (1 I ts)
  then have min (j' - 1) (j-1) = i by auto
  with 1 show ?case by auto
next
  case (3 I v v' t)
  then have min (j' - 1) (j-1) = i by (auto simp: list_all2_Cons2 upt_eq_Cons_conv)
  with 3 show ?case by auto
next
  case (4 I x xs t t' ts)
  from 4(1)[of tl ys xs' ts' Suc i] 4(2-6) show ?case
  by (auto simp add: list_all2_Cons2 upt_eq_Cons_conv Suc_less_eq2
    elim!: list_rel_mono_strong_split: prod.splits if_splits)
qed simp

lemma in_foldr_UnI: x ∈ A ⇒ A ∈ set xs ⇒ x ∈ foldr (∪) xs {}
  by (induction xs) auto

lemma in_foldr_UnE: x ∈ foldr (∪) xs {} ⇒ (∧A. A ∈ set xs ⇒ x ∈ A ⇒ P) ⇒ P
  by (induction xs) auto

lemma sat_the_restrict: fv φ ⊆ A ⇒ MFOTL.sat σ (map the (restrict A v)) i φ = MFOTL.sat σ
  (map the v) i φ
  by (rule sat_fvi_cong) (auto intro!: map_the_restrict)

lemma update_since:
  assumes pre: wf_since_aux σ n R pos φ I ψ aux ne
  and qtable1: qtable n (MFOTL.fv φ) (mem_restr R) (λv. MFOTL.sat σ (map the v) ne φ) rel1
  and qtable2: qtable n (MFOTL.fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) ne ψ) rel2
  and result_eq: (rel, aux') = update_since I pos rel1 rel2 (τ σ ne) aux
  and fvi_subset: MFOTL.fv φ ⊆ MFOTL.fv ψ
  shows wf_since_aux σ n R pos φ I ψ aux' (Suc ne)
  and qtable n (MFOTL.fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) ne (Sincep pos φ I ψ))
rel
proof -
  let ?wf_tuple = λv. wf_tuple n (MFOTL.fv ψ) v
  note sat.simps[simp del]

  define aux0 where aux0 = [(t, join rel pos rel1). (t, rel) ← aux, τ σ ne - t ≤ right I]
  have sorted_aux0: sorted_wrt (λx y. fst x > fst y) aux0
  using pre[unfolded wf_since_aux_def, THEN conjunct1]
  unfolding aux0_def
  by (induction aux) (auto simp add: sorted_wrt_append)
  have in_aux0_1: (t, X) ∈ set aux0 ⇒ ne ≠ 0 ∧ t ≤ τ σ (ne-1) ∧ τ σ ne - t ≤ right I ∧
  (∃ i. τ σ i = t) ∧
  qtable n (MFOTL.fv ψ) (mem_restr R) (λv. (MFOTL.sat σ (map the v) (ne-1) (Sincep pos φ (point
  (τ σ (ne-1) - t)) ψ) ∧
  (if pos then MFOTL.sat σ (map the v) ne φ else ¬ MFOTL.sat σ (map the v) ne φ))) X for t X
  unfolding aux0_def using fvi_subset
  by (auto 0 1 elim!: qtable_join[OF qtable1] simp: sat_the_restrict
    dest!: assms(1)[unfolded wf_since_aux_def, THEN conjunct2, THEN conjunct1, rule_format])
  then have in_aux0_le_τ: (t, X) ∈ set aux0 ⇒ t ≤ τ σ ne for t X
  by (meson τ_mono diff_le_self le_trans)
  have in_aux0_2: ne ≠ 0 ⇒ t ≤ τ σ (ne-1) ⇒ τ σ ne - t ≤ right I ⇒ ∃ i. τ σ i = t ⇒
  ∃ X. (t, X) ∈ set aux0 for t
  proof -

```

```

fix t
assume ne ≠ 0 t ≤ τ σ (ne-1) τ σ ne - t ≤ right I ∃ i. τ σ i = t
then obtain X where (t, X) ∈ set aux
  by (atomize_elim, intro assms(1)[unfolded wf_since_aux_def, THEN conjunct2, THEN conjunct2,
rule_format])
    (auto simp: gr0_conv_Suc elim!: order_trans[rotated] intro!: diff_le_mono τ_mono)
with ⟨τ σ ne - t ≤ right I⟩ have (t, join X pos rel1) ∈ set aux0
  unfolding aux0_def by (auto elim!: beX[rotated] intro!: exI[of _ X])
then show ∃ X. (t, X) ∈ set aux0
  by blast
qed
have aux0_Nil: aux0 = [] ⇒ ne = 0 ∨ ne ≠ 0 ∧ (∀ t. t ≤ τ σ (ne-1) ∧ τ σ ne - t ≤ right I →
  (∃ i. τ σ i = t))
  using in_aux0_2 by (cases ne = 0) (auto)

have aux'_eq: aux' = (case aux0 of
  [] ⇒ [(τ σ ne, rel2)]
  | x # aux' ⇒ (if fst x = τ σ ne then (fst x, snd x ∪ rel2) # aux' else (τ σ ne, rel2) # x # aux'))
  using result_eq unfolding aux0_def update_since_def Let_def by simp
have sorted_aux': sorted_wrt (λx y. fst x > fst y) aux'
  unfolding aux'_eq
  using sorted_aux0 in_aux0_le_τ by (cases aux0) (fastforce)+
have in_aux'_1: t ≤ τ σ ne ∧ τ σ ne - t ≤ right I ∧ (∃ i. τ σ i = t) ∧
  qtable n (MFOTL.fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) ne (Sincep pos φ (point (τ
σ ne - t)) ψ)) X
  if aux': (t, X) ∈ set aux' for t X
proof (cases aux0)
  case Nil
  with aux' show ?thesis
  unfolding aux'_eq using qtable2 aux0_Nil
  by (auto simp: zero_enat_def[symmetric] sat_Since_rec[where i=ne]
  dest: spec[of _ τ σ (ne-1)] elim!: qtable_cong[OF _ refl])
next
case (Cons a as)
show ?thesis
proof (cases t = τ σ ne)
  case t: True
  show ?thesis
  proof (cases fst a = τ σ ne)
  case True
  with aux' Cons t have X = snd a ∪ rel2
  unfolding aux'_eq using sorted_aux0 by auto
  moreover from in_aux0_1[of fst a snd a] Cons have ne ≠ 0
  fst a ≤ τ σ (ne - 1) τ σ ne - fst a ≤ right I ∃ i. τ σ i = fst a
  qtable n (fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) (ne - 1)
  (Sincep pos φ (point (τ σ (ne - 1) - fst a)) ψ) ∧ (if pos then MFOTL.sat σ (map the v) ne φ
  else ¬ MFOTL.sat σ (map the v) ne φ)) (snd a)
  by auto
  ultimately show ?thesis using qtable2 t True
  by (auto simp: sat_Since_rec[where i=ne] sat.simps(3) elim!: qtable_union)
next
case False
  with aux' Cons t have X = rel2
  unfolding aux'_eq using sorted_aux0 in_aux0_le_τ[of fst a snd a] by auto
  with aux' Cons t False show ?thesis
  unfolding aux'_eq using qtable2 in_aux0_2[of τ σ (ne-1)] in_aux0_le_τ[of fst a snd a]
sorted_aux0
  by (auto simp: sat_Since_rec[where i=ne] sat.simps(3) zero_enat_def[symmetric] enat_0_iff

```

```

not_le
  elim!: qtable_cong[OF _ refl] dest!: le_τ_less meta_mp)
qed
next
  case False
  with aux' Cons have (t, X) ∈ set aux0
  unfolding aux'_eq by (auto split: if_splits)
  then have ne ≠ 0 t ≤ τ σ (ne - 1) τ σ ne - t ≤ right I ∃ i. τ σ i = t
  qtable n (fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) (ne - 1) (Since pos φ (point (τ
σ (ne - 1) - t)) ψ) ∧
  (if pos then MFOTL.sat σ (map the v) ne φ else ¬ MFOTL.sat σ (map the v) ne φ)) X
  using in_aux0_1 by blast+
  with False aux' Cons show ?thesis
  unfolding aux'_eq using qtable2
  by (fastforce simp: sat_Since_rec[where i=ne] sat_simps(3)
  diff_diff_right[where i=τ σ ne and j=τ σ _ + τ σ ne and k=τ σ (ne - 1),
  OF trans_le_add2, simplified] elim!: qtable_cong[OF _ refl] order_trans dest: le_τ_less)
qed
qed

have in_aux'_2: ∃ X. (t, X) ∈ set aux' if t ≤ τ σ ne τ σ ne - t ≤ right I ∃ i. τ σ i = t for t
proof (cases t = τ σ ne)
  case True
  then show ?thesis
  proof (cases aux0)
    case Nil
    with True show ?thesis unfolding aux'_eq by simp
  next
    case (Cons a as)
    with True show ?thesis unfolding aux'_eq
    by (cases fst a = τ σ ne) auto
  qed
next
  case False
  with that have ne ≠ 0
  using le_τ_less neq0_conv by blast
  moreover from False that have t ≤ τ σ (ne-1)
  by (metis One_nat_def Suc_leI Suc_pred τ_mono diff_is_0_eq' order.antisym neq0_conv not_le)
  ultimately obtain X where (t, X) ∈ set aux0 using ⟨τ σ ne - t ≤ right I⟩ ⟨∃ i. τ σ i = t⟩
  by atomize_elim (auto intro!: in_aux0_2)
  then show ?thesis unfolding aux'_eq using False
  by (auto intro: exI[of _ X] split: list.split)
qed

show wf_since_aux σ n R pos φ I ψ aux' (Suc ne)
  unfolding wf_since_aux_def
  by (auto dest: in_aux'_1 intro: sorted_aux' in_aux'_2)

have rel_eq: rel = foldr (∪) [rel. (t, rel) ← aux', left I ≤ τ σ ne - t] {}
  unfolding aux'_eq aux0_def
  using result_eq by (simp add: update_since_def Let_def)
have rel_alt: rel = (∪ (t, rel) ∈ set aux'. if left I ≤ τ σ ne - t then rel else empty_table)
  unfolding rel_eq
  by (auto elim!: in_foldr_UnE bexI[rotated] intro!: in_foldr_UnI)
show qtable n (fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) ne (Since pos φ I ψ)) rel
  unfolding rel_alt
proof (rule qtable_Union[where Qi=λ(t, X) v.
  left I ≤ τ σ ne - t ∧ MFOTL.sat σ (map the v) ne (Since pos φ (point (τ σ ne - t)) ψ)],

```

```

goal_cases finite qtable equiv)
case (equiv v)
show ?case
proof (rule iffI, erule sat_Since_point, goal_cases left right)
  case (left j)
  then show ?case using in_aux'_2[of  $\tau$   $\sigma$  j, OF __ exI, OF __ refl] by auto
next
  case right
  then show ?case by (auto elim!: sat_Since_pointD dest: in_aux'_1)
qed
qed (auto dest!: in_aux'_1 intro!: qtable_empty)
qed

lemma length_update_until: length (update_until pos I rel1 rel2 nt aux) = Suc (length aux)
unfolding update_until_def by simp

lemma wf_update_until:
assumes pre: wf_until_aux  $\sigma$  n R pos  $\varphi$  I  $\psi$  aux ne
  and qtable1: qtable n (MFOTL.fv  $\varphi$ ) (mem_restr R) ( $\lambda v$ . MFOTL.sat  $\sigma$  (map the v) (ne + length aux)  $\varphi$ ) rel1
  and qtable2: qtable n (MFOTL.fv  $\psi$ ) (mem_restr R) ( $\lambda v$ . MFOTL.sat  $\sigma$  (map the v) (ne + length aux)  $\psi$ ) rel2
  and fvi_subset: MFOTL.fv  $\varphi$   $\subseteq$  MFOTL.fv  $\psi$ 
shows wf_until_aux  $\sigma$  n R pos  $\varphi$  I  $\psi$  (update_until I pos rel1 rel2 ( $\tau$   $\sigma$  (ne + length aux)) aux) ne
unfolding wf_until_aux_def length_update_until
unfolding update_until_def list.rel_map add_Suc_right upt.simps eqTrueI[OF le_add1] if_True
proof (rule list_all2_appendI, unfold list.rel_map, goal_cases old new)
  case old
  show ?case
  proof (rule list.rel_mono_strong[OF assms(1)[unfolded wf_until_aux_def]]); safe, goal_cases mono1 mono2)
    case (mono1 i X Y v)
    then show ?case
    by (fastforce simp: sat_the_restrict less_Suc_eq
      elim!: qtable_join[OF __ qtable1] qtable_union[OF __ qtable1])
  next
    case (mono2 i X Y v)
    then show ?case using fvi_subset
    by (auto 0  $\exists$  simp: sat_the_restrict less_Suc_eq split: if_splits
      elim!: qtable_union[OF __ qtable_join_fixed[OF qtable2]]
      elim: qtable_cong[OF __ refl] intro: exI[of __ ne + length aux])
  qed
next
  case new
  then show ?case
  by (auto intro!: qtable_empty qtable1 qtable2[THEN qtable_cong] exI[of __ ne + length aux]
    simp: less_Suc_eq zero_enat_def[symmetric])
qed

lemma wf_until_aux_Cons: wf_until_aux  $\sigma$  n R pos  $\varphi$  I  $\psi$  (a # aux) ne  $\implies$ 
wf_until_aux  $\sigma$  n R pos  $\varphi$  I  $\psi$  aux (Suc ne)
unfolding wf_until_aux_def
by (simp add: upt_conv_Cons del: upt_Suc cong: if_cong)

lemma wf_until_aux_Cons1: wf_until_aux  $\sigma$  n R pos  $\varphi$  I  $\psi$  ((t, a1, a2) # aux) ne  $\implies$  t =  $\tau$   $\sigma$  ne
unfolding wf_until_aux_def
by (simp add: upt_conv_Cons del: upt_Suc)

```

**lemma** *wf\_until\_aux\_Cons3*:  $wf\_until\_aux\ \sigma\ n\ R\ pos\ \varphi\ I\ \psi\ ((t, a1, a2) \# aux)\ ne \implies$   
 $qtable\ n\ (MFOTL.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v. (\exists j. ne \leq j \wedge j < Suc\ (ne + length\ aux) \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ ne)\ I \wedge$   
 $MFOTL.sat\ \sigma\ (map\ the\ v)\ j\ \psi \wedge (\forall k \in \{ne..<j\}. if\ pos\ then\ MFOTL.sat\ \sigma\ (map\ the\ v)\ k\ \varphi\ else\ \neg MFOTL.sat\ \sigma\ (map\ the\ v)\ k\ \varphi)))\ a2$   
**unfolding** *wf\_until\_aux\_def*  
**by** (*simp add: upt\_conv\_Cons del: upt\_Suc*)

**lemma** *upt\_append*:  $a \leq b \implies b \leq c \implies [a..<b] @ [b..<c] = [a..<c]$   
**by** (*metis le\_Suc\_ex upt\_add\_eq\_append*)

**lemma** *wf\_mbuf2\_add*:  
**assumes** *wf\_mbuf2 i ja jb P Q buf*  
**and** *list\_all2 P [ja..<ja'] xs*  
**and** *list\_all2 Q [jb..<jb'] ys*  
**and**  $ja \leq ja'\ jb \leq jb'$   
**shows** *wf\_mbuf2 i ja' jb' P Q (mbuf2\_add xs ys buf)*  
**using** *assms unfolding wf\_mbuf2\_def*  
**by** (*auto 0 3 simp: list\_all2\_append2 upt\_append dest: list\_all2\_lengthD*  
*intro: exI[where x=[i..<ja]] exI[where x=[ja..<ja']]*  
*exI[where x=[i..<jb]] exI[where x=[jb..<jb']] split: prod.splits*)

**lemma** *mbuf2\_take\_eqD*:  
**assumes** *mbuf2\_take f buf = (xs, buf')*  
**and** *wf\_mbuf2 i ja jb P Q buf*  
**shows** *wf\_mbuf2 (min ja jb) ja jb P Q buf'*  
**and** *list\_all2 ( $\lambda i z. \exists x y. P\ i\ x \wedge Q\ i\ y \wedge z = f\ x\ y$ ) [i..<min ja jb] xs*  
**using** *assms unfolding wf\_mbuf2\_def*  
**by** (*induction f buf arbitrary: i xs buf' rule: mbuf2\_take.induct*)  
*(fastforce simp add: list\_all2\_Cons2 upt\_eq\_Cons\_conv min\_absorb1 min\_absorb2 split: prod.splits)+*

**lemma** *mbuf2t\_take\_eqD*:  
**assumes** *mbuf2t\_take f z buf nts = (z', buf', nts')*  
**and** *wf\_mbuf2 i ja jb P Q buf*  
**and** *list\_all2 R [i..<j] nts*  
**and**  $ja \leq j\ jb \leq j$   
**shows** *wf\_mbuf2 (min ja jb) ja jb P Q buf'*  
**and** *list\_all2 R [min ja jb..<j] nts'*  
**using** *assms unfolding wf\_mbuf2\_def*  
**by** (*induction f z buf nts arbitrary: i z' buf' nts' rule: mbuf2t\_take.induct*)  
*(auto simp add: list\_all2\_Cons2 upt\_eq\_Cons\_conv less\_eq\_Suc\_le min\_absorb1 min\_absorb2*  
*split: prod.split)*

**lemma** *mbuf2t\_take\_induct[consumes 5, case\_names base step]*:  
**assumes** *mbuf2t\_take f z buf nts = (z', buf', nts')*  
**and** *wf\_mbuf2 i ja jb P Q buf*  
**and** *list\_all2 R [i..<j] nts*  
**and**  $ja \leq j\ jb \leq j$   
**and**  $U\ i\ z$   
**and**  $\bigwedge k\ x\ y\ t\ z. i \leq k \implies Suc\ k \leq ja \implies Suc\ k \leq jb \implies$   
 $P\ k\ x \implies Q\ k\ y \implies R\ k\ t \implies U\ k\ z \implies U\ (Suc\ k)\ (f\ x\ y\ t\ z)$   
**shows**  $U\ (min\ ja\ jb)\ z'$   
**using** *assms unfolding wf\_mbuf2\_def*  
**by** (*induction f z buf nts arbitrary: i z' buf' nts' rule: mbuf2t\_take.induct*)  
*(auto simp add: list\_all2\_Cons2 upt\_eq\_Cons\_conv less\_eq\_Suc\_le min\_absorb1 min\_absorb2*  
*elim!: arg\_cong2[of \_ \_ \_ \_ U, OF \_ refl, THEN iffD1, rotated] split: prod.split)*

**lemma** *mbuf2\_take\_add'*:

**assumes**  $eq: mbuf2\_take\ f\ (mbuf2\_add\ xs\ ys\ buf) = (zs, buf')$   
**and**  $pre: wf\_mbuf2'\ \sigma\ j\ n\ R\ \varphi\ \psi\ buf$   
**and**  $xs: list\_all2\ (\lambda i. qtable\ n\ (MFOTL.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi))$   
 $[progress\ \sigma\ \varphi\ j..<progress\ \sigma\ \varphi\ j']\ xs$   
**and**  $ys: list\_all2\ (\lambda i. qtable\ n\ (MFOTL.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \psi))$   
 $[progress\ \sigma\ \psi\ j..<progress\ \sigma\ \psi\ j']\ ys$   
**and**  $j \leq j'$   
**shows**  $wf\_mbuf2'\ \sigma\ j'\ n\ R\ \varphi\ \psi\ buf'$   
**and**  $list\_all2\ (\lambda i\ Z. \exists X\ Y.$   
 $qtable\ n\ (MFOTL.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi)\ X \wedge$   
 $qtable\ n\ (MFOTL.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \psi)\ Y \wedge$   
 $Z = f\ X\ Y)$   
 $[min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)..<min\ (progress\ \sigma\ \varphi\ j')\ (progress\ \sigma\ \psi\ j')]\ zs$   
**using**  $pre$  **unfolding**  $wf\_mbuf2'\_def$   
**by**  $(force\ intro!:\ mbuf2\_take\_eqD[OF\ eq]\ wf\_mbuf2\_add[OF\ \_]\ xs\ ys]\ progress\_mono[OF\ \langle j \leq j' \rangle])+$

**lemma**  $mbuf2t\_take\_add'$ :

**assumes**  $eq: mbuf2t\_take\ f\ z\ (mbuf2\_add\ xs\ ys\ buf)\ nts = (z', buf', nts')$   
**and**  $pre\_buf: wf\_mbuf2'\ \sigma\ j\ n\ R\ \varphi\ \psi\ buf$   
**and**  $pre\_nts: list\_all2\ (\lambda i\ t. t = \tau\ \sigma\ i)\ [min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)..<j']\ nts$   
**and**  $xs: list\_all2\ (\lambda i. qtable\ n\ (MFOTL.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi))$   
 $[progress\ \sigma\ \varphi\ j..<progress\ \sigma\ \varphi\ j']\ xs$   
**and**  $ys: list\_all2\ (\lambda i. qtable\ n\ (MFOTL.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \psi))$   
 $[progress\ \sigma\ \psi\ j..<progress\ \sigma\ \psi\ j']\ ys$   
**and**  $j \leq j'$   
**shows**  $wf\_mbuf2'\ \sigma\ j'\ n\ R\ \varphi\ \psi\ buf'$   
**and**  $wf\_ts\ \sigma\ j'\ \varphi\ \psi\ nts'$   
**using**  $pre\_buf\ pre\_nts$  **unfolding**  $wf\_mbuf2'\_def\ wf\_ts\_def$   
**by**  $(blast\ intro!:\ mbuf2t\_take\_eqD[OF\ eq]\ wf\_mbuf2\_add[OF\ \_]\ xs\ ys]\ progress\_mono[OF\ \langle j \leq j' \rangle]\ progress\_le)+$

**lemma**  $mbuf2t\_take\_add\_induct'$ [consumes 6, case\_names base step]:

**assumes**  $eq: mbuf2t\_take\ f\ z\ (mbuf2\_add\ xs\ ys\ buf)\ nts = (z', buf', nts')$   
**and**  $pre\_buf: wf\_mbuf2'\ \sigma\ j\ n\ R\ \varphi\ \psi\ buf$   
**and**  $pre\_nts: list\_all2\ (\lambda i\ t. t = \tau\ \sigma\ i)\ [min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)..<j']\ nts$   
**and**  $xs: list\_all2\ (\lambda i. qtable\ n\ (MFOTL.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi))$   
 $[progress\ \sigma\ \varphi\ j..<progress\ \sigma\ \varphi\ j']\ xs$   
**and**  $ys: list\_all2\ (\lambda i. qtable\ n\ (MFOTL.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \psi))$   
 $[progress\ \sigma\ \psi\ j..<progress\ \sigma\ \psi\ j']\ ys$   
**and**  $j \leq j'$   
**and**  $base: U\ (min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j))\ z$   
**and**  $step: \bigwedge k\ X\ Y\ z. min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j) \leq k \implies$   
 $Suc\ k \leq progress\ \sigma\ \varphi\ j' \implies Suc\ k \leq progress\ \sigma\ \psi\ j' \implies$   
 $qtable\ n\ (MFOTL.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ k\ \varphi)\ X \implies$   
 $qtable\ n\ (MFOTL.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ k\ \psi)\ Y \implies$   
 $U\ k\ z \implies U\ (Suc\ k)\ (f\ X\ Y\ (\tau\ \sigma\ k)\ z)$   
**shows**  $U\ (min\ (progress\ \sigma\ \varphi\ j')\ (progress\ \sigma\ \psi\ j'))\ z'$   
**using**  $pre\_buf\ pre\_nts$  **unfolding**  $wf\_mbuf2'\_def\ wf\_ts\_def$   
**by**  $(blast\ intro!:\ mbuf2t\_take\_induct[OF\ eq]\ wf\_mbuf2\_add[OF\ \_]\ xs\ ys]\ progress\_mono[OF\ \langle j \leq j' \rangle]\ progress\_le\ base\ step)$

**lemma**  $progress\_Until\_le$ :  $progress\ \sigma\ (formula.Until\ \varphi\ I\ \psi)\ j \leq min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)$   
**by**  $(cases\ right\ I)\ (auto\ simp:\ trans\_le\_add1\ intro!:\ cInf\_lower)$

**lemma**  $list\_all2\_upt\_Cons$ :  $P\ a\ x \implies list\_all2\ P\ [Suc\ a..<b]\ xs \implies Suc\ a \leq b \implies$   
 $list\_all2\ P\ [a..<b]\ (x\ \# \ xs)$   
**by**  $(simp\ add:\ list\_all2\_Cons2\ upt\_eq\_Cons\_conv)$

**lemma** *list\_all2\_upt\_append*:  $list\_all2\ P\ [a..<b]\ xs \implies list\_all2\ P\ [b..<c]\ ys \implies$   
 $a \leq b \implies b \leq c \implies list\_all2\ P\ [a..<c]\ (xs\ @\ ys)$   
**by** (*induction xs arbitrary: a*) (*auto simp add: list\_all2\_Cons2\_upt\_eq\_Cons\_conv*)

**lemma** *meval*:

**assumes** *wf\_mformula*  $\sigma\ j\ n\ R\ \varphi\ \varphi'$   
**shows** *case meval*  $n\ (\tau\ \sigma\ j)\ (\Gamma\ \sigma\ j)\ \varphi$  *of*  $(xs, \varphi_n) \Rightarrow wf\_mformula\ \sigma\ (Suc\ j)\ n\ R\ \varphi_n\ \varphi' \wedge$   
 $list\_all2\ (\lambda i. qtable\ n\ (MFOTL.fv\ \varphi')\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi'))$   
 $[progress\ \sigma\ \varphi'\ j..<progress\ \sigma\ \varphi'\ (Suc\ j)]\ xs$   
**using** *assms proof* (*induction*  $\varphi$  *arbitrary: n R*  $\varphi'$ )  
**case** (*MRel rel*)  
**then show** *?case*  
**by** (*cases pred: wf\_mformula*)  
*(auto simp add: ball\_Un intro: wf\_mformula.intros qtableI table\_eq\_rel eq\_rel\_eval\_trm*  
*in\_eq\_rel qtable\_empty qtable\_unit\_table)*  
**next**  
**case** (*MPred e ts*)  
**then show** *?case*  
**by** (*cases pred: wf\_mformula*)  
*(auto 0 4 simp: table\_def in\_these\_eq match\_wf\_tuple match\_eval\_trm image\_iff dest: ex\_match*  
*split: if\_splits intro!: wf\_mformula.intros qtableI elim!: bexI[rotated])*  
**next**  
**case** (*MAnd*  $\varphi\ pos\ \psi\ buf$ )  
**from** *MAnd.prem*s **show** *?case*  
**by** (*cases pred: wf\_mformula*)  
*(auto simp: fvi\_And sat\_And fvi\_And\_Not sat\_And\_Not sat\_the\_restrict*  
*dest!: MAnd.IH split: if\_splits prod\_splits intro!: wf\_mformula.And qtable\_join*  
*dest: mbuf2\_take\_add' elim!: list.rel\_mono\_strong)*  
**next**  
**case** (*MOr*  $\varphi\ \psi\ buf$ )  
**from** *MOr.prem*s **show** *?case*  
**by** (*cases pred: wf\_mformula*)  
*(auto dest!: MOr.IH split: if\_splits prod\_splits intro!: wf\_mformula.Or qtable\_union*  
*dest: mbuf2\_take\_add' elim!: list.rel\_mono\_strong)*  
**next**  
**case** (*MExists*  $\varphi$ )  
**from** *MExists.prem*s **show** *?case*  
**by** (*cases pred: wf\_mformula*)  
*(force simp: list.rel\_map fvi\_Suc sat\_fvi\_cong nth\_Cons'*  
*intro!: wf\_mformula.Exists dest!: MExists.IH qtable\_project\_fv*  
*elim!: list.rel\_mono\_strong table\_fvi\_tl qtable\_cong sat\_fvi\_cong[THEN iffD1, rotated -1]*  
*split: if\_splits)+*  
**next**  
**case** (*MPrev*  $I\ \varphi\ first\ buf\ nts$ )  
**from** *MPrev.prem*s **show** *?case*  
**proof** (*cases pred: wf\_mformula*)  
**case** (*Prev*  $\psi$ )  
**let**  $?xs = fst\ (meval\ n\ (\tau\ \sigma\ j)\ (\Gamma\ \sigma\ j)\ \varphi)$   
**let**  $?\varphi = snd\ (meval\ n\ (\tau\ \sigma\ j)\ (\Gamma\ \sigma\ j)\ \varphi)$   
**let**  $?ls = fst\ (mprev\_next\ I\ (buf\ @\ ?xs)\ (nts\ @\ [\tau\ \sigma\ j]))$   
**let**  $?rs = fst\ (snd\ (mprev\_next\ I\ (buf\ @\ ?xs)\ (nts\ @\ [\tau\ \sigma\ j])))$   
**let**  $?ts = snd\ (snd\ (mprev\_next\ I\ (buf\ @\ ?xs)\ (nts\ @\ [\tau\ \sigma\ j])))$   
**let**  $?P = \lambda i\ X. qtable\ n\ (fv\ \psi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \psi)\ X$   
**let**  $?min = min\ (progress\ \sigma\ \psi\ (Suc\ j))\ (Suc\ j - 1)$   
**from** *Prev MPrev.IH*[*of*  $n\ R\ \psi$ ] **have** *IH*:  $wf\_mformula\ \sigma\ (Suc\ j)\ n\ R\ ?\varphi\ \psi$  **and**  
 $list\_all2\ ?P\ [progress\ \sigma\ \psi\ j..<progress\ \sigma\ \psi\ (Suc\ j)]\ ?xs$  **by** *auto*  
**with** *Prev*(4,5) **have**  $list\_all2\ (\lambda i\ X. if\ mem\ (\tau\ \sigma\ (Suc\ i)) - \tau\ \sigma\ i\ I\ then\ ?P\ i\ X\ else\ X = empty\_table)$   
 $[min\ (progress\ \sigma\ \psi\ j)\ (j - 1)..<?min)]\ ?ls \wedge$

```

list_all2 ?P [?min..<progress σ ψ (Suc j)] ?rs ∧
list_all2 (λi t. t = τ σ i) [?min..<Suc j] ?ts
by (intro mprev)
(auto simp: progress_mono progress_le simp del:
intro!: list_all2_upt_append list_all2_appendI order.trans[OF min.cobounded1])
moreover have min (Suc (progress σ ψ j)) j = Suc (min (progress σ ψ j) (j-1)) if j > 0
using that by auto
ultimately show ?thesis using progress_mono[of j Suc j σ ψ] Prev(1,3) IH
by (auto simp: map_Suc_upt[symmetric] upt_Suc[of 0] list.rel_map qtable_empty_iff
simp del: upt_Suc elim!: wf_mformula.Prev list.rel_mono_strong
split: prod.split if_split_asm)
qed simp
next
case (MNext I φ first nts)
from MNext.premis show ?case
proof (cases pred: wf_mformula)
case (Next ψ)

have min[simp]:
min (progress σ ψ j - Suc 0) (j - Suc 0) = progress σ ψ j - Suc 0
min (progress σ ψ (Suc j) - Suc 0) j = progress σ ψ (Suc j) - Suc 0 for j
using progress_le[of σ ψ j] progress_le[of σ ψ Suc j] by auto

let ?xs = fst (meval n (τ σ j) (Γ σ j) φ)
let ?ys = case (?xs, first) of (_ # xs, True) ⇒ xs | _ ⇒ ?xs
let ?φ = snd (meval n (τ σ j) (Γ σ j) φ)
let ?ls = fst (mprev_next I ?ys (nts @ [τ σ j]))
let ?rs = fst (snd (mprev_next I ?ys (nts @ [τ σ j])))
let ?ts = snd (snd (mprev_next I ?ys (nts @ [τ σ j])))
let ?P = λi X. qtable n (fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) i ψ) X
let ?min = min (progress σ ψ (Suc j) - 1) (Suc j - 1)
from Next MNext.IH[of n R ψ] have IH: wf_mformula σ (Suc j) n R ?φ ψ
list_all2 ?P [progress σ ψ j..<progress σ ψ (Suc j)] ?xs by auto
with Next have list_all2 (λi X. if mem (τ σ (Suc i)) - τ σ i I then ?P (Suc i) X else X =
empty_table)
[progress σ ψ j - 1..<?min] ?ls ∧
list_all2 ?P [Suc ?min..<progress σ ψ (Suc j)] ?rs ∧
list_all2 (λi t. t = τ σ i) [?min..<Suc j] ?ts if progress σ ψ j < progress σ ψ (Suc j)
using progress_le[of σ ψ j] that
by (intro mnext)
(auto simp: progress_mono list_all2_Cons2 upt_eq_Cons_conv
intro!: list_all2_upt_append list_all2_appendI split: list.splits)
then show ?thesis using progress_mono[of j Suc j σ ψ] progress_le[of σ ψ Suc j] Next IH
by (cases progress σ ψ (Suc j) > progress σ ψ j)
(auto 0 3 simp: qtable_empty_iff le_Suc_eq le_diff_conv
elim!: wf_mformula.Next list.rel_mono_strong list_all2_appendI
split: prod.split list.splits if_split_asm)
qed simp
next
case (MSince pos φ I ψ buf nts aux)
note sat.simps[simp del]
from MSince.premis obtain φ'' φ''' ψ'' where Since_eq: φ' = MFOTL.Since φ''' I ψ''
and pos: if pos then φ''' = φ'' else φ''' = MFOTL.Neg φ''
and pos_eq: safe_formula φ''' = pos
and φ: wf_mformula σ j n R φ φ''
and ψ: wf_mformula σ j n R ψ ψ''
and fvi_subset: MFOTL.fv φ'' ⊆ MFOTL.fv ψ''
and buf: wf_mbuf2' σ j n R φ'' ψ'' buf

```

```

and nts: wf_ts σ j φ'' ψ'' nts
and aux: wf_since_aux σ n R pos φ'' I ψ'' aux (progress σ (formula.Since φ''' I ψ'') j)
by (cases pred: wf_mformula) (auto)
have φ''': MFOTL.fv φ''' = MFOTL.fv φ'' progress σ φ''' j = progress σ φ'' j for j
using pos by (simp_all split: if_splits)
have nts_snoc: list_all2 (λi t. t = τ σ i)
  [min (progress σ φ'' j) (progress σ ψ'' j)..<Suc j] (nts @ [τ σ j])
using nts unfolding wf_ts_def
by (auto simp add: progress_le[THEN min.coboundedI1] intro: list_all2_appendI)
have update: wf_since_aux σ n R pos φ'' I ψ'' (snd (zs, aux')) (progress σ (formula.Since φ''' I ψ'')
(Suc j)) ∧
  list_all2 (λi. qtable n (MFOTL.fv φ''' ∪ MFOTL.fv ψ'') (mem_restr R)
    (λv. MFOTL.sat σ (map the v) i (formula.Since φ''' I ψ'')))
  [progress σ (formula.Since φ''' I ψ'') j..<progress σ (formula.Since φ''' I ψ'') (Suc j)] (fst (zs, aux'))
if eval_φ: fst (meval n (τ σ j) (Γ σ j) φ) = xs
  and eval_ψ: fst (meval n (τ σ j) (Γ σ j) ψ) = ys
  and eq: mbuf2t_take (λr1 r2 t (zs, aux)).
    case update_since I pos r1 r2 t aux of (z, x) ⇒ (zs @ [z], x)
    ([], aux) (mbuf2t_add xs ys buf) (nts @ [τ σ j]) = ((zs, aux'), buf', nts')
for xs ys zs aux' buf' nts'
unfolding progress.simps φ'''
proof (rule mbuf2t_take_add_induct'[where j'=Suc j and z'=(zs, aux'), OF eq buf nts_snoc],
  goal_cases xs ys _ base step)
  case xs
  then show ?case
  using MSince.IH(1)[OF φ] eval_φ by auto
next
  case ys
  then show ?case
  using MSince.IH(2)[OF ψ] eval_ψ by auto
next
  case base
  then show ?case
  using aux by (simp add: φ''')
next
  case (step k X Y z)
  then show ?case
  using fvi_subset pos
  by (auto simp: Un_absorb1 elim!: update_since(1) list_all2_appendI dest!: update_since(2)
    split: prod.split if_splits)
qed simp
with MSince.IH(1)[OF φ] MSince.IH(2)[OF ψ] show ?case
by (auto 0 3 simp: Since_eq split: prod.split
  intro: wf_mformula.Since[OF _ _ pos pos_eq fvi_subset]
  elim: mbuf2t_take_add'(1)[OF _ buf nts_snoc] mbuf2t_take_add'(2)[OF _ buf nts_snoc])
next
  case (MUntil pos φ I ψ buf nts aux)
  note sat.simps[simp del] progress.simps[simp del]
  from MUntil.prem obtain φ'' φ''' ψ'' where Until_eq: φ' = MFOTL.Until φ''' I ψ''
  and pos: if pos then φ''' = φ'' else φ''' = MFOTL.Neg φ''
  and pos_eq: safe_formula φ''' = pos
  and φ: wf_mformula σ j n R φ φ''
  and ψ: wf_mformula σ j n R ψ ψ''
  and fvi_subset: MFOTL.fv φ'' ⊆ MFOTL.fv ψ''
  and buf: wf_mbuf2t' σ j n R φ'' ψ'' buf
  and nts: wf_ts σ j φ'' ψ'' nts
  and aux: wf_until_aux σ n R pos φ'' I ψ'' aux (progress σ (formula.Until φ''' I ψ'') j)
  and length_aux: progress σ (formula.Until φ''' I ψ'') j + length aux =

```

```

    min (progress  $\sigma$   $\varphi''$   $j$ ) (progress  $\sigma$   $\psi''$   $j$ )
  by (cases pred: wf_mformula) (auto)
have  $\varphi'''$ : progress  $\sigma$   $\varphi'''$   $j$  = progress  $\sigma$   $\varphi''$   $j$  for  $j$ 
  using pos by (simp_all add: progress.simps split: if_splits)
have nts_snoc: list_all2 ( $\lambda i$   $t$ .  $t = \tau \sigma i$ )
  [ $\min$  (progress  $\sigma$   $\varphi''$   $j$ ) (progress  $\sigma$   $\psi''$   $j$ ).. $\text{Suc } j$ ] (nts @ [ $\tau \sigma j$ ])
  using nts unfolding wf_ts_def
  by (auto simp add: progress_le[THEN min.coboundedI1] intro: list_all2_appendI)
{
fix xs ys zs aux' aux'' buf' nts'
assume eval_φ: fst (meval  $n$  ( $\tau \sigma j$ ) ( $\Gamma \sigma j$ )  $\varphi$ ) = xs
  and eval_ψ: fst (meval  $n$  ( $\tau \sigma j$ ) ( $\Gamma \sigma j$ )  $\psi$ ) = ys
  and eq1: mbuf2t_take (update_until  $I$  pos) aux (mbuf2_add xs ys buf) (nts @ [ $\tau \sigma j$ ]) =
    (aux', buf', nts')
  and eq2: eval_until  $I$  (case nts' of []  $\Rightarrow \tau \sigma j$  | nt # _  $\Rightarrow nt$ ) aux' = (zs, aux'')
have update1: wf_until_aux  $\sigma$   $n$   $R$  pos  $\varphi''$   $I$   $\psi''$  aux' (progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ )  $j$ )  $\wedge$ 
  progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ )  $j$  + length aux' =
  min (progress  $\sigma$   $\varphi'''$  ( $\text{Suc } j$ )) (progress  $\sigma$   $\psi''$  ( $\text{Suc } j$ ))
  using MUntil.IH(1)[OF  $\varphi$ ] eval_φ MUntil.IH(2)[OF  $\psi$ ]
  eval_ψ nts_snoc nts_snoc length_aux aux fvi_subset
  unfolding  $\varphi'''$ 
  by (elim mbuf2t_take_add_induct'[where  $j' = \text{Suc } j$ , OF eq1 buf])
  (auto simp: length_update_until elim: wf_update_until)
have nts': wf_ts  $\sigma$  ( $\text{Suc } j$ )  $\varphi''$   $\psi''$  nts'
  using MUntil.IH(1)[OF  $\varphi$ ] eval_φ MUntil.IH(2)[OF  $\psi$ ] eval_ψ nts_snoc
  unfolding wf_ts_def
  by (intro mbuf2t_take_eqD(2)[OF eq1]) (auto simp: progress_mono progress_le
    intro: wf_mbuf2_add buf[unfolded wf_mbuf2'_def])
have  $i \leq$  progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ ) ( $\text{Suc } j$ )  $\implies$ 
  wf_until_aux  $\sigma$   $n$   $R$  pos  $\varphi''$   $I$   $\psi''$  aux'  $i \implies$ 
   $i +$  length aux' = min (progress  $\sigma$   $\varphi'''$  ( $\text{Suc } j$ )) (progress  $\sigma$   $\psi''$  ( $\text{Suc } j$ ))  $\implies$ 
  wf_until_aux  $\sigma$   $n$   $R$  pos  $\varphi''$   $I$   $\psi''$  aux'' (progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ ) ( $\text{Suc } j$ ))  $\wedge$ 
   $i +$  length zs = progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ ) ( $\text{Suc } j$ )  $\wedge$ 
   $i +$  length zs + length aux'' = min (progress  $\sigma$   $\varphi'''$  ( $\text{Suc } j$ )) (progress  $\sigma$   $\psi''$  ( $\text{Suc } j$ ))  $\wedge$ 
  list_all2 ( $\lambda i$ . qtable  $n$  (MFOTL.fv  $\psi''$ ) (mem_restr  $R$ )
    ( $\lambda v$ . MFOTL.sat  $\sigma$  (map the  $v$ )  $i$  (formula.Until (if pos then  $\varphi''$  else MFOTL.Neg  $\varphi''$ )  $I$   $\psi''$ )))
  [ $i$ .. $i +$  length zs] zs for  $i$ 
  using eq2
proof (induction aux' arbitrary: zs aux''  $i$ )
  case Nil
  then show ?case by (auto dest!: antisym[OF progress_Until_le])
next
  case (Cons a aux')
  obtain  $t$   $a1$   $a2$  where  $a = (t, a1, a2)$  by (cases a)
  from Cons.prem1(2) have aux': wf_until_aux  $\sigma$   $n$   $R$  pos  $\varphi''$   $I$   $\psi''$  aux' ( $\text{Suc } i$ )
    by (rule wf_until_aux_Cons)
  from Cons.prem1(2) have 1:  $t = \tau \sigma i$ 
    unfolding  $\langle a = (t, a1, a2) \rangle$  by (rule wf_until_aux_Cons1)
  from Cons.prem1(2) have 3: qtable  $n$  (MFOTL.fv  $\psi''$ ) (mem_restr  $R$ ) ( $\lambda v$ .
    ( $\exists j \geq i$ .  $j < \text{Suc } (i + \text{length } aux') \wedge \text{mem } (\tau \sigma j - \tau \sigma i) I \wedge \text{MFOTL.sat } \sigma$  (map the  $v$ )  $j$   $\psi'' \wedge$ 
    ( $\forall k \in \{i..<j\}$ . if pos then MFOTL.sat  $\sigma$  (map the  $v$ )  $k$   $\varphi''$  else  $\neg \text{MFOTL.sat } \sigma$  (map the  $v$ )  $k$   $\varphi''$ )))
a2
    unfolding  $\langle a = (t, a1, a2) \rangle$  by (rule wf_until_aux_Cons3)
  from Cons.prem1(3) have Suc_i_aux':  $\text{Suc } i +$  length aux' =
    min (progress  $\sigma$   $\varphi'''$  ( $\text{Suc } j$ )) (progress  $\sigma$   $\psi''$  ( $\text{Suc } j$ ))
    by simp
  have  $i \geq$  progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ ) ( $\text{Suc } j$ )
    if enat (case nts' of []  $\Rightarrow \tau \sigma j$  | nt #  $x \Rightarrow nt$ )  $\leq$  enat  $t +$  right  $I$ 

```

```

using that nts' unfolding wf_ts_def progress.simps
by (auto simp add: 1 list_all2_Cons2 upt_eq_Cons_conv  $\varphi'''$ 
  intro!: cInf_lower  $\tau$ _mono elim!: order.trans[rotated] simp del: upt_Suc split: if_splits list.splits)
moreover
have Suc  $i \leq$  progress  $\sigma$  (formula.Until  $\varphi'''$  I  $\psi''$ ) (Suc j)
  if enat  $t +$  right I < enat (case nts' of []  $\Rightarrow$   $\tau \sigma j$  | nt #  $x \Rightarrow$  nt)
proof -
  from that obtain m where m: right I = enat m by (cases right I) auto
  have  $\tau$ _min:  $\tau \sigma$  (min j k) = min ( $\tau \sigma j$ ) ( $\tau \sigma k$ ) for k
    by (simp add: min_of_mono monoI)
  have le_progress_iff[simp]:  $i \leq$  progress  $\sigma \varphi i \iff$  progress  $\sigma \varphi i = i$  for  $\varphi i$ 
    using progress_le[of  $\sigma \varphi i$ ] by auto
  have min_Suc[simp]: min j (Suc j) = j by auto
  let ?X = {i.  $\forall k. k <$  Suc j  $\wedge k \leq$  min (progress  $\sigma \varphi'''$  (Suc j)) (progress  $\sigma \psi''$  (Suc j))  $\longrightarrow$  enat
    ( $\tau \sigma k$ )  $\leq$  enat ( $\tau \sigma i$ ) + right I}
  let ?min = min j (min (progress  $\sigma \varphi''$  (Suc j)) (progress  $\sigma \psi''$  (Suc j)))
  have  $\tau \sigma$  ?min  $\leq$   $\tau \sigma j$ 
    by (rule  $\tau$ _mono) auto
  from m have ?X  $\neq$  {}
    by (auto dest!:  $\tau$ _mono[of _ progress  $\sigma \varphi''$  (Suc j)  $\sigma$ ]
      simp: not_le not_less  $\varphi'''$  intro!: exI[of _ progress  $\sigma \varphi''$  (Suc j)])
  thm less_le_trans[of  $\tau \sigma i + m$   $\tau \sigma$  _  $\tau \sigma$  _ + m]
  from m show ?thesis
    using that nts' unfolding wf_ts_def progress.simps
    by (intro cInf_greatest[OF  $\langle ?X \neq \{ \} \rangle$ ]
      (auto simp: 1  $\varphi'''$  not_le not_less list_all2_Cons2 upt_eq_Cons_conv less_Suc_eq
        simp del: upt_Suc split: list.splits if_splits
        dest!: spec[of _ ?min] less_le_trans[of  $\tau \sigma i + m$   $\tau \sigma$  _  $\tau \sigma$  _ + m] less_τD)
qed
moreover have *:  $k <$  progress  $\sigma \psi$  (Suc j) if
  enat ( $\tau \sigma i$ ) + right I < enat (case nts' of []  $\Rightarrow$   $\tau \sigma j$  | nt #  $x \Rightarrow$  nt)
  enat ( $\tau \sigma k - \tau \sigma i$ )  $\leq$  right I  $\psi = \psi'' \vee \psi = \varphi''$  for k  $\psi$ 
proof -
  from that(1,2) obtain m where right I = enat m
     $\tau \sigma i + m <$  (case nts' of []  $\Rightarrow$   $\tau \sigma j$  | nt #  $x \Rightarrow$  nt)  $\tau \sigma k - \tau \sigma i \leq m$ 
    by (cases right I) auto
  with that(3) nts' progress_le[of  $\sigma \psi''$  Suc j] progress_le[of  $\sigma \varphi''$  Suc j]
  show ?thesis
    unfolding wf_ts_def le_diff_conv
    by (auto simp: not_le list_all2_Cons2 upt_eq_Cons_conv less_Suc_eq add commute
      simp del: upt_Suc split: list.splits if_splits dest!: le_less_trans[of  $\tau \sigma k$ ] less_τD)
qed
ultimately show ?case using Cons.premS Suc_i_aux'[simplified]
  unfolding  $\langle a = (t, a1, a2) \rangle$ 
  by (auto simp:  $\varphi'''$  1 sat.simps upt_conv_Cons dest!: Cons.IH[OF _ aux' Suc_i_aux']
    simp del: upt_Suc split: if_splits prod.splits intro!: iff_exI qtable_cong[OF 3 refl])
qed
note this[OF progress_mono[OF le_SucI, OF order.refl] conjunct1[OF update1] conjunct2[OF update1]]
}
note update = this
from MUntil.IH(1)[OF  $\varphi$ ] MUntil.IH(2)[OF  $\psi$ ] pos_pos_eq fwi_subset show ?case
  by (auto 0 4 simp: Until_eq  $\varphi'''$  progress.simps(3) split: prod.split if_splits
    dest!: update[OF refl refl, rotated]
    intro!: wf_mformula.Until
    elim!: list.rel_mono_strong qtable_cong
    elim: mbuf2t_take_add'(1)[OF _ buf_nts_snoc] mbuf2t_take_add'(2)[OF _ buf_nts_snoc])
qed

```

#### 6.5.4 Monitor step

**lemma** *wf\_mstate\_mstep*:  $wf\_mstate\ \varphi\ \pi\ R\ st \implies last\_ts\ \pi \leq snd\ tdb \implies wf\_mstate\ \varphi\ (psnoc\ \pi\ tdb)\ R\ (snd\ (mstep\ tdb\ st))$

**unfolding** *wf\_mstate\_def mstep\_def Let\_def*

**by** (*fastforce simp add: progress\_mono le\_imp\_diff\_is\_add split: prod.splits elim!: prefix\_of\_psnocE dest: meval\_list\_all2\_lengthD*)

**lemma** *mstep\_output\_iff*:

**assumes**  $wf\_mstate\ \varphi\ \pi\ R\ st\ last\_ts\ \pi \leq snd\ tdb\ prefix\_of\ (psnoc\ \pi\ tdb)\ \sigma\ mem\_restr\ R\ v$

**shows**  $(i, v) \in fst\ (mstep\ tdb\ st) \iff$

$progress\ \sigma\ \varphi\ (plen\ \pi) \leq i \wedge i < progress\ \sigma\ \varphi\ (Suc\ (plen\ \pi)) \wedge$

$wf\_tuple\ (MFOTL.nfv\ \varphi)\ (MFOTL.fv\ \varphi)\ v \wedge MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi$

**proof** –

**from** *prefix\_of\_psnocE[OF assms(3,2)]* **have** *prefix\_of*  $\pi\ \sigma$

$\Gamma\ \sigma\ (plen\ \pi) = fst\ tdb\ \tau\ \sigma\ (plen\ \pi) = snd\ tdb$  **by** *auto*

**moreover from** *assms(1) <prefix\_of*  $\pi\ \sigma$  **have** *mstate\_n st = MFOTL.nfv*  $\varphi$

*mstate\_i st = progress*  $\sigma\ \varphi\ (plen\ \pi)\ wf\_mformula\ \sigma\ (plen\ \pi)\ (mstate\_n\ st)\ R\ (mstate\_m\ st)\ \varphi$

**unfolding** *wf\_mstate\_def* **by** *blast+*

**moreover from** *meval[OF <wf\_mformula*  $\sigma\ (plen\ \pi)\ (mstate\_n\ st)\ R\ (mstate\_m\ st)\ \varphi$ ]

**obtain**  $Vs$  *st'* **where**

*meval*  $(mstate\_n\ st)\ (\tau\ \sigma\ (plen\ \pi))\ (\Gamma\ \sigma\ (plen\ \pi))\ (mstate\_m\ st) = (Vs, st')$

*wf\_mformula*  $\sigma\ (Suc\ (plen\ \pi))\ (mstate\_n\ st)\ R\ st'\ \varphi$

*list\_all2*  $(\lambda i. qtable\ (mstate\_n\ st)\ (fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi))$

*[progress*  $\sigma\ \varphi\ (plen\ \pi) \leq i < progress\ \sigma\ \varphi\ (Suc\ (plen\ \pi))]$   $Vs$  **by** *blast*

**moreover from** *this assms(4)* **have** *qtable*  $(mstate\_n\ st)\ (fv\ \varphi)\ (mem\_restr\ R)$

$(\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi)\ (Vs!\ (i - progress\ \sigma\ \varphi\ (plen\ \pi)))$

**if** *progress*  $\sigma\ \varphi\ (plen\ \pi) \leq i < progress\ \sigma\ \varphi\ (Suc\ (plen\ \pi))$

**using** *that* **by** (*auto simp: list\_all2\_conv\_all\_nth*

*dest!: spec[of \_ (i - progress*  $\sigma\ \varphi\ (plen\ \pi))]$ )

**ultimately show** *?thesis*

**using** *assms(4) unfolding mstep\_def Let\_def*

**by** (*auto simp: in\_set\_enumerate\_eq list\_all2\_conv\_all\_nth progress\_mono le\_imp\_diff\_is\_add*

*elim!: in\_qtableE in\_qtableI intro!: bexI[of \_ (i, Vs!\ (i - progress*  $\sigma\ \varphi\ (plen\ \pi))]$ )

**qed**

#### 6.5.5 Monitor function

**definition** *minit\_safe* **where**

*minit\_safe*  $\varphi = (if\ mmonitorable\_exec\ \varphi\ then\ minit\ \varphi\ else\ undefined)$

**lemma** *minit\_safe\_minit*:  $mmonitorable\ \varphi \implies minit\_safe\ \varphi = minit\ \varphi$

**unfolding** *minit\_safe\_def monitorable\_formula\_code* **by** *simp*

**lemma** (**in** *monitorable\_mfotl*) *mstep\_mverdicts*:

**assumes** *wf*:  $wf\_mstate\ \varphi\ \pi\ R\ st$

**and** *le[simp]*:  $last\_ts\ \pi \leq snd\ tdb$

**and** *restrict*:  $mem\_restr\ R\ v$

**shows**  $(i, v) \in fst\ (mstep\ tdb\ st) \iff (i, v) \in M\ (psnoc\ \pi\ tdb) - M\ \pi$

**proof** –

**obtain**  $\sigma$  **where** *p2*: *prefix\_of*  $(psnoc\ \pi\ tdb)\ \sigma$

**using** *ex\_prefix\_of* **by** *blast*

**with** *le* **have** *p1*: *prefix\_of*  $\pi\ \sigma$  **by** (*blast elim!: prefix\_of\_psnocE*)

**show** *?thesis*

**unfolding** *M\_def*

**by** (*auto 0 3 simp: p2 progress\_prefix\_conv[OF \_ p1] sat\_prefix\_conv[OF \_ p1] not\_less*

*pprogress\_eq[OF p1] pprogress\_eq[OF p2]*

*dest: mstep\_output\_iff[OF wf le p2 restrict, THEN iffD1] spec[of \_ \sigma]*

*mstep\_output\_iff[OF wf le \_ restrict, THEN iffD1] progress\_sat\_cong[OF p1]*)

intro: mstep\_output\_iff[OF wf le p2 restrict, THEN iffD2] p1)

qed

**primrec** msteps0 **where**  
 msteps0 [] st = ({}, st)  
 | msteps0 (tdb # π) st =  
 (let (V', st') = mstep tdb st; (V'', st'') = msteps0 π st' in (V' ∪ V'', st'))

**primrec** msteps0\_stateless **where**  
 msteps0\_stateless [] st = {}  
 | msteps0\_stateless (tdb # π) st = (let (V', st') = mstep tdb st in V' ∪ msteps0\_stateless π st')

**lemma** msteps0\_msteps0\_stateless: fst (msteps0 w st) = msteps0\_stateless w st  
 by (induct w arbitrary: st) (auto simp: split\_beta)

**lift\_definition** msteps :: 'a MFOTL.prefix ⇒ 'a mstate ⇒ (nat × 'a option list) set × 'a mstate  
 is msteps0 .

**lift\_definition** msteps\_stateless :: 'a MFOTL.prefix ⇒ 'a mstate ⇒ (nat × 'a option list) set  
 is msteps0\_stateless .

**lemma** msteps\_msteps\_stateless: fst (msteps w st) = msteps\_stateless w st  
 by transfer (rule msteps0\_msteps0\_stateless)

**lemma** msteps0\_snoc: msteps0 (π @ [tdb]) st =  
 (let (V', st') = msteps0 π st; (V'', st'') = mstep tdb st' in (V' ∪ V'', st'))  
 by (induct π arbitrary: st) (auto split: prod.splits)

**lemma** msteps\_psnoc: last\_ts π ≤ snd tdb ⇒ msteps (psnoc π tdb) st =  
 (let (V', st') = msteps π st; (V'', st'') = mstep tdb st' in (V' ∪ V'', st'))  
 by transfer (auto simp: msteps0\_snoc split: list.splits prod.splits if\_splits)

**definition** monitor **where**  
 monitor φ π = msteps\_stateless π (minit\_safe φ)

**lemma** Suc\_length\_conv\_snoc: (Suc n = length xs) = (∃ y ys. xs = ys @ [y] ∧ length ys = n)  
 by (cases xs rule: rev\_cases) auto

**lemma** (in monitorable\_mfotl) wf\_mstate\_msteps: wf\_mstate φ π R st ⇒ mem\_restr R v ⇒ π ≤ π' ⇒  
 X = msteps (pdrop (plen π) π') st ⇒ wf\_mstate φ π' R (snd X) ∧  
 ((i, v) ∈ fst X) = ((i, v) ∈ M π' - M π)

**proof** (induct plen π' - plen π arbitrary: X st π π')

case 0  
**from** 0(1,4,5) **have** π = π' X = ({}, st)  
**by** (transfer; auto)+  
**with** 0(2) **show** ?case **by** simp

**next**  
**case** (Suc x)  
**from** Suc(2,5) **obtain** π'' tdb **where** x = plen π'' - plen π π ≤ π''  
 π' = psnoc π'' tdb pdrop (plen π) (psnoc π'' tdb) = psnoc (pdrop (plen π) π') tdb  
 last\_ts (pdrop (plen π) π') ≤ snd tdb last\_ts π'' ≤ snd tdb  
 π'' ≤ psnoc π'' tdb

**proof** (atomize\_elim, transfer, elim exE, goal\_cases prefix)  
**case** (prefix \_\_ π' \_ π\_tdb)  
**then show** ?case  
**proof** (cases π\_tdb rule: rev\_cases)  
**case** (snoc π tdb)

```

with prefix show ?thesis
  by (intro beXI[of _ π' @ π] exI[of _ tdb])
      (force simp: sorted_append append_eq_Cons_conv split: list_splits if_splits)+
qed simp
qed
with Suc(1)[OF this(1) Suc.prem(1,2) this(2) refl] Suc.prem show ?case
  unfolding msteps_msteps_stateless[symmetric]
  by (auto simp: msteps_psnoc split_beta mstep_mverdicts
      dest: mono_monitor[THEN set_mp, rotated] intro!: wf_mstate_mstep)
qed

lemma (in monitorable_mfotl) wf_mstate_msteps_stateless:
  assumes wf_mstate φ π R st mem_restr R v π ≤ π'
  shows (i, v) ∈ msteps_stateless (pdrop (plen π) π') st  $\longleftrightarrow$  (i, v) ∈ M π' - M π
  using wf_mstate_msteps[OF assms refl] unfolding msteps_msteps_stateless by simp

lemma (in monitorable_mfotl) wf_mstate_msteps_stateless_UNIV: wf_mstate φ π UNIV st  $\implies$  π ≤ π'  $\implies$ 
  msteps_stateless (pdrop (plen π) π') st = M π' - M π
  by (auto dest: wf_mstate_msteps_stateless[OF _ mem_restr_UNIV])

lemma (in monitorable_mfotl) mverdicts_Nil: M pnil = {}
  by (simp add: M_def pprogress_eq)

lemma wf_mstate_minit_safe: mmonitorable φ  $\implies$  wf_mstate φ pnil R (minit_safe φ)
  using wf_mstate_minit_minit_safe_minit mmonitorable_def by metis

lemma (in monitorable_mfotl) monitor_mverdicts: monitor φ π = M π
  unfolding monitor_def using monitorable
  by (subst wf_mstate_msteps_stateless_UNIV[OF wf_mstate_minit_safe, simplified])
      (auto simp: mmonitorable_def mverdicts_Nil)

```

## 6.6 Collected correctness results

**context** monitorable\_mfotl  
**begin**

We summarize the main results proved above.

1. The term  $M$  describes semantically the monitor's expected behaviour:
  - *mono\_monitor*:  $\pi \leq \pi' \implies M \pi \subseteq M \pi'$
  - *sound\_monitor*:  $\llbracket (i, v) \in M \pi; \text{prefix\_of } \pi \sigma \rrbracket \implies \text{MFOTL.sat } \sigma (\text{map the } v) i \varphi$
  - *complete\_monitor*:  $\llbracket \text{prefix\_of } \pi \sigma; \text{wf\_tuple } (\text{MFOTL.nfv } \varphi) (\text{fv } \varphi) v; \bigwedge \sigma. \text{prefix\_of } \pi \sigma \implies \text{MFOTL.sat } \sigma (\text{map the } v) i \varphi \rrbracket \implies \exists \pi'. \text{prefix\_of } \pi' \sigma \wedge (i, v) \in M \pi'$
  - *sliceable\_M*:  $\text{mem\_restr } S v \implies ((i, v) \in M (\text{pmap\_}\Gamma (\lambda D. D \cap \text{relevant\_events } \varphi S) \pi)) = ((i, v) \in M \pi)$
2. The executable monitor's online interface *minit\_safe* and *mstep* preserves the invariant *wf\_mstate* and produces the the verdicts according to  $M$ :
  - *wf\_mstate\_minit\_safe*:  $\text{mmonitorable } \varphi' \implies \text{wf\_mstate } \varphi' \text{ pnil } R (\text{minit\_safe } \varphi')$
  - *wf\_mstate\_mstep*:  $\llbracket \text{wf\_mstate } \varphi' \pi R \text{ st}; \text{last\_ts } \pi \leq \text{snd tdb} \rrbracket \implies \text{wf\_mstate } \varphi' (\text{psnoc } \pi \text{ tdb}) R (\text{snd } (\text{mstep tdb st}))$
  - *mstep\_mverdicts*:  $\llbracket \text{wf\_mstate } \varphi \pi R \text{ st}; \text{last\_ts } \pi \leq \text{snd tdb}; \text{mem\_restr } R v \rrbracket \implies ((i, v) \in \text{fst } (\text{mstep tdb st})) = ((i, v) \in M (\text{psnoc } \pi \text{ tdb}) - M \pi)$

3. The executable monitor's offline interface *Monitor.monitor* implements *M*:

- *monitor\_mverdicts*: *Monitor.monitor*  $\varphi$   $\pi = M$   $\pi$

end

## 7 Slicing framework

This section formalizes the abstract slicing framework and the joint data slicer presented in the article [3, Sections 4.2 and 4.3].

### 7.1 Abstract slicing

#### 7.1.1 Definition 1

Corresponds to locale *monitor* defined in theory *MFOTL\_Monitor.Abstract\_Monitor*.

#### 7.1.2 Definition 2

```

locale slicer = monitor +
  fixes submonitor :: 'k :: finite  $\Rightarrow$  'a prefix  $\Rightarrow$  (nat  $\times$  'b option list) set
  and splitter :: 'a prefix  $\Rightarrow$  'k  $\Rightarrow$  'a prefix
  and joiner :: ('k  $\Rightarrow$  (nat  $\times$  'b option list) set)  $\Rightarrow$  (nat  $\times$  'b option list) set
assumes mono_splitter:  $\pi \leq \pi' \Longrightarrow$  splitter  $\pi$  k  $\leq$  splitter  $\pi'$  k
  and correct_slicer: joiner ( $\lambda k$ . submonitor k (splitter  $\pi$  k)) = M  $\pi$ 
begin

```

```

lemmas sound_slicer = equalityD1[OF correct_slicer]
lemmas complete_slicer = equalityD2[OF correct_slicer]

```

end

```

locale self_slicer = slicer nfv fv sat M  $\lambda$ _. M splitter joiner for nfv fv sat M splitter joiner

```

#### 7.1.3 Definition 3

```

locale event_separable_splitter =
  fixes event_splitter :: 'a  $\Rightarrow$  'k :: finite set
begin

lift_definition splitter :: 'a prefix  $\Rightarrow$  'k  $\Rightarrow$  'a prefix is
   $\lambda \pi$  k. map ( $\lambda(D, t)$ . ( $\{e \in D. k \in \text{event\_splitter } e\}, t$ ))  $\pi$ 
  by (auto simp: o_def split_beta)

```

#### 7.1.4 Lemma 1

```

lemma mono_splitter:  $\pi \leq \pi' \Longrightarrow$  splitter  $\pi$  k  $\leq$  splitter  $\pi'$  k
  by transfer auto

```

end

## 7.2 Joint data slicer

```

abbreviation (input) ok  $\varphi$  v  $\equiv$  wf_tuple (MFOTL.nfv  $\varphi$ ) (MFOTL.fv  $\varphi$ ) v

```

```

locale splitting_strategy =

```

```

fixes  $\varphi :: 'a$  MFOTL.formula
and  $strategy :: 'a$  option list  $\Rightarrow 'k ::$  finite set
assumes  $strategy\_nonempty: ok \varphi v \Longrightarrow strategy v \neq \{\}$ 
begin

abbreviation  $slice\_set$  where
   $slice\_set k \equiv \{v. \exists v'. map\ the\ v' = v \wedge ok \varphi v' \wedge k \in strategy\ v'\}$ 

end

```

### 7.2.1 Definition 4

```

locale  $MFOTL\_monitor =$ 
   $monitor\ MFOTL.nfv\ \varphi\ MFOTL.fv\ \varphi\ \lambda\sigma\ v\ i. MFOTL.sat\ \sigma\ v\ i\ \varphi\ M$  for  $\varphi\ M$ 

locale  $joint\_data\_slicer = MFOTL\_monitor\ \varphi\ M + splitting\_strategy\ \varphi\ strategy$ 
for  $\varphi\ M\ strategy$ 
begin

definition  $event\_splitter$  where
   $event\_splitter\ e = (\bigcup (strategy\ ' \{v. ok\ \varphi\ v \wedge MFOTL.matches\ (map\ the\ v)\ \varphi\ e\}))$ 

```

```

sublocale  $event\_separable\_splitter$  where  $event\_splitter = event\_splitter$  .

```

```

definition  $joiner$  where
   $joiner = (\lambda s. \bigcup k. s\ k \cap (UNIV :: nat\ set) \times \{v. k \in strategy\ v\})$ 

```

```

lemma  $splitter\_pslice: splitter\ \pi\ k = MFOTL\_slicer.pslice\ \varphi\ (slice\_set\ k)\ \pi$ 
by  $transfer\ (auto\ simp: event\_splitter\_def)$ 

```

### 7.2.2 Lemma 2

Corresponds to the following theorem  $sat\_slice\_strong$  proved in theory  $MFOTL\_Monitor.Abstract\_Monitor$ :

$$\llbracket relevant\_events\ \varphi'\ S \subseteq E; v \in S \rrbracket \Longrightarrow MFOTL.sat\ \sigma\ v\ i\ \varphi' = MFOTL.sat\ (map\_ \Gamma\ (\lambda D. D \cap E)\ \sigma)\ v\ i\ \varphi'$$

### 7.2.3 Theorem 1

```

sublocale  $joint\_monitor: MFOTL\_monitor\ \varphi\ \lambda\pi. joiner\ (\lambda k. M\ (splitter\ \pi\ k))$ 
proof  $(unfold\ locales, goal\_cases\ mono\ wf\ sound\ complete)$ 
  case  $(mono\ \pi\ \pi')$ 
  show  $?case$ 
  using  $mono\_monitor[OF\ mono\_splitter, OF\ mono]$ 
  by  $(auto\ simp: joiner\_def)$ 
next
  case  $(wf\ i\ v\ \pi)$ 
  then obtain  $k$  where  $in\_M: (i, v) \in M\ (splitter\ \pi\ k)$  and  $k: k \in strategy\ v$ 
  unfolding  $joiner\_def$  by  $(auto\ split: if\_splits)$ 
  then show  $?case$ 
  using  $wf\_monitor[OF\ in\_M]$  by  $auto$ 
next
  case  $(sound\ i\ v\ \pi\ \sigma)$ 
  then obtain  $k$  where  $in\_M: (i, v) \in M\ (splitter\ \pi\ k)$  and  $k: k \in strategy\ v$ 
  unfolding  $joiner\_def$  by  $(auto\ split: if\_splits)$ 
  have  $wf: ok\ \varphi\ v$  and  $sat: \bigwedge \sigma. prefix\_of\ (splitter\ \pi\ k)\ \sigma \Longrightarrow MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi$ 
  using  $sound\_monitor[OF\ in\_M]\ wf\_monitor[OF\ in\_M]$  by  $auto$ 
  then have  $MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi$  if  $prefix\_of\ \pi\ \sigma$  for  $\sigma$ 
  using  $that\ k$ 

```

```

  by (intro iffD2[OF sat_slice_iff[of map the v slice_set k σ i φ]])
    (auto simp: wf_tuple_def fvi_less_nfv splitter_pslice intro!: exI[of _ v] prefix_of_pmap_Γ)
  then show ?case using sound(2) by blast
next
case (complete π σ v i)
with strategy_nonempty obtain k where k: k ∈ strategy v by blast
have MFOTL.sat σ' (map the v) i φ if prefix_of (MFOTL_slicer.pslice φ (slice_set k) π) σ' for σ'
proof -
  have MFOTL.sat σ' (map the v) i φ = MFOTL.sat (MFOTL_slicer.slice φ (slice_set k) σ') (map
the v) i φ
  using complete(2) k by (auto intro!: sat_slice_iff)
  also have ... = MFOTL.sat (MFOTL_slicer.slice φ (slice_set k) (replace_prefix π σ')) (map the v)
i φ
  using that complete k by (subst slice_replace_prefix[symmetric]; simp)
  also have ... = MFOTL.sat (replace_prefix π σ') (map the v) i φ
  using complete(2) k by (auto intro!: sat_slice_iff[symmetric])
  also have ...
  by (rule complete(3)[rule_format], rule prefix_of_replace_prefix[OF that])
  finally show ?thesis .
qed
with complete(1-3) obtain π' where π':
prefix_of π' (MFOTL_slicer.slice φ (slice_set k) σ) (i, v) ∈ M π'
by (atomize_elim, intro complete_monitor[where π=MFOTL_slicer.pslice φ (slice_set k) π])
(auto simp: splitter_pslice intro!: prefix_of_pmap_Γ)
from π'(1) obtain π'' where π' = MFOTL_slicer.pslice φ (slice_set k) π'' prefix_of π'' σ
by (atomize_elim, rule prefix_of_map_Γ_D)
with π' k show ?case
by (intro exI[of _ π'']) (auto simp: joiner_def splitter_pslice intro!: exI[of _ k])
qed

```

#### 7.2.4 Corollary 1

```

sublocale joint_slicer: slicer MFOTL.nfv φ MFOTL.fv φ λσ v i. MFOTL.sat σ v i φ
λπ. joiner (λk. M (splitter π k)) λ_. M splitter joiner
by standard (auto simp: mono_splitter)

```

end

#### 7.2.5 Definition 5

Corresponds to locale *sliceable\_monitor* defined in theory *MFOTL\_Monitor.Abstract\_Monitor*.

```

locale slicable_joint_data_slicer =
sliceable_monitor MFOTL.nfv φ MFOTL.fv φ relevant_events φ λσ v i. MFOTL.sat σ v i φ M +
joint_data_slicer φ M strategy for φ M strategy
begin

```

```

lemma monitor_split: ok φ v ⇒ k ∈ strategy v ⇒ (i, v) ∈ M (splitter π k) ↔ (i, v) ∈ M π
unfolding splitter_pslice
by (rule sliceable_M)
(auto simp: wf_tuple_def fvi_less_nfv intro!: mem_restrI[rotated 2, where y=map the v])

```

#### 7.2.6 Theorem 2

```

sublocale self_slicer MFOTL.nfv φ MFOTL.fv φ λσ v i. MFOTL.sat σ v i φ M splitter joiner
proof (standard, erule mono_splitter, safe, goal_cases sound complete)
case (sound π i v)
have ok φ v using joint_monitor.wf_monitor[OF sound] by auto
from sound obtain k where (i, v) ∈ M (splitter π k) k ∈ strategy v

```

```

    unfolding joiner_def by blast
  with ⟨ok φ v⟩ show ?case by (simp add: monitor_split)
next
  case (complete π i v)
  have ok φ v using wf_monitor[OF complete] by auto
  with complete_strategy_nonempty obtain k where k: k ∈ strategy v by blast
  then have (i, v) ∈ M (splitter π k) using complete ⟨ok φ v⟩ by (simp add: monitor_split)
  with k show ?case unfolding joiner_def by blast
qed

end

```

### 7.2.7 Towards Theorem 3

**fun** names :: 'a MFOTL.formula ⇒ MFOTL.name set **where**

```

  names (MFOTL.Pred e _) = {e}
| names (MFOTL.Eq _ _) = {}
| names (MFOTL.Neg ψ) = names ψ
| names (MFOTL.Or α β) = names α ∪ names β
| names (MFOTL.Exists ψ) = names ψ
| names (MFOTL.Prev I ψ) = names ψ
| names (MFOTL.Next I ψ) = names ψ
| names (MFOTL.Since α I β) = names α ∪ names β
| names (MFOTL.Until α I β) = names α ∪ names β

```

**fun** gen\_unique :: 'a MFOTL.formula ⇒ bool **where**

```

  gen_unique (MFOTL.Pred _ _) = True
| gen_unique (MFOTL.Eq (MFOTL.Var _) (MFOTL.Const _)) = False
| gen_unique (MFOTL.Eq (MFOTL.Const _) (MFOTL.Var _)) = False
| gen_unique (MFOTL.Eq _ _) = True
| gen_unique (MFOTL.Neg ψ) = gen_unique ψ
| gen_unique (MFOTL.Or α β) = (gen_unique α ∧ gen_unique β ∧ names α ∩ names β = {})
| gen_unique (MFOTL.Exists ψ) = gen_unique ψ
| gen_unique (MFOTL.Prev I ψ) = gen_unique ψ
| gen_unique (MFOTL.Next I ψ) = gen_unique ψ
| gen_unique (MFOTL.Since α I β) = (gen_unique α ∧ gen_unique β ∧ names α ∩ names β = {})
| gen_unique (MFOTL.Until α I β) = (gen_unique α ∧ gen_unique β ∧ names α ∩ names β = {})

```

**lemma** sat\_inter\_names\_cong:  $(\bigwedge e. e \in \text{names } \varphi \implies \{xs. (e, xs) \in E\} = \{xs. (e, xs) \in F\}) \implies$   
 $\text{MFOTL.sat } (\text{map\_}\Gamma (\lambda D. D \cap E) \sigma) v i \varphi \longleftrightarrow \text{MFOTL.sat } (\text{map\_}\Gamma (\lambda D. D \cap F) \sigma) v i \varphi$   
**by** (induction φ arbitrary: v i) (auto split: nat.splits)

**lemma** matches\_in\_names:  $\text{MFOTL.matches } v \varphi x \implies \text{fst } x \in \text{names } \varphi$   
**by** (induction φ arbitrary: v) (auto)

**lemma** unique\_names\_matches\_absorb:  $\text{fst } x \in \text{names } \alpha \implies \text{names } \alpha \cap \text{names } \beta = \{\} \implies$   
 $\text{MFOTL.matches } v \alpha x \vee \text{MFOTL.matches } v \beta x \longleftrightarrow \text{MFOTL.matches } v \alpha x$   
 $\text{fst } x \in \text{names } \beta \implies \text{names } \alpha \cap \text{names } \beta = \{\} \implies$   
 $\text{MFOTL.matches } v \alpha x \vee \text{MFOTL.matches } v \beta x \longleftrightarrow \text{MFOTL.matches } v \beta x$   
**by** (auto dest: matches\_in\_names)

**definition** mergeable\_envs **where**

```

mergeable_envs n S ⟷ (∀ v1 ∈ S. ∀ v2 ∈ S. (∀ A B f.
  (∀ x ∈ A. x < n ∧ v1 ! x = f x) ∧ (∀ x ∈ B. x < n ∧ v2 ! x = f x) ⟶
  (∃ v ∈ S. ∀ x ∈ A ∪ B. v ! x = f x)))

```

**lemma** mergeable\_envsI:

```

assumes ⋀ v1 v2 v. v1 ∈ S ⟹ v2 ∈ S ⟹ length v = n ⟹ ∀ x < n. v ! x = v1 ! x ∨ v ! x = v2 ! x
⟹ v ∈ S

```

**shows** *mergeable\_envs* *n S*  
**unfolding** *mergeable\_envs\_def*  
**proof** (*safe, goal\_cases mergeable*)  
**case** [*simp*]: (*mergeable v1 v2 A B f*)  
**let** *?v* = *tabulate* ( $\lambda x. \text{if } x \in A \cup B \text{ then } f \ x \text{ else } v1 \ ! \ x$ ) 0 *n*  
**from** *assms*[*of v1 v2 ?v, simplified*] **show** *?case*  
**by** (*auto intro!*: *beXI[of \_ ?v]*)  
**qed**

**lemma** *in\_listset\_nth*:  $x \in \text{listset } As \implies i < \text{length } As \implies x \ ! \ i \in As \ ! \ i$   
**by** (*induction As arbitrary: x i*) (*auto simp: set\_Cons\_def nth\_Cons split: nat.split*)

**lemma** *all\_nth\_in\_listset*:  $\text{length } x = \text{length } As \implies (\bigwedge i. i < \text{length } As \implies x \ ! \ i \in As \ ! \ i) \implies x \in \text{listset } As$   
**by** (*induction x As rule: list\_induct2*) (*fastforce simp: set\_Cons\_def nth\_Cons*)+

**lemma** *mergeable\_envs\_listset*: *mergeable\_envs* (*length As*) (*listset As*)  
**by** (*rule mergeable\_envsI*) (*auto intro!: all\_nth\_in\_listset elim!: in\_listset\_nth*)

**lemma** *mergeable\_envs\_Ex*: *mergeable\_envs n S*  $\implies \text{MFOTL.nfv } \alpha \leq n \implies \text{MFOTL.nfv } \beta \leq n \implies$   
 $(\exists v' \in S. \forall x \in \text{fv } \alpha. v' \ ! \ x = v \ ! \ x) \implies (\exists v' \in S. \forall x \in \text{fv } \beta. v' \ ! \ x = v \ ! \ x) \implies$   
 $(\exists v' \in S. \forall x \in \text{fv } \alpha \cup \text{fv } \beta. v' \ ! \ x = v \ ! \ x)$   
**proof** (*clarify, goal\_cases mergeable*)  
**case** (*mergeable v1 v2*)  
**then show** *?case*  
**by** (*auto intro: order.strict\_trans2[OF fvi\_less\_nfv[rule\_format]]*)  
*elim!*: *mergeable\_envs\_def*[*THEN iffD1, rule\_format, of \_ \_ v1 v2*])  
**qed**

**lemma** *in\_set\_ConsE*:  $xs \in \text{set\_Cons } A \ As \implies (\bigwedge y \ \text{ys}. xs = y \ \# \ \text{ys} \implies y \in A \implies \text{ys} \in As \implies P) \implies P$   
**unfolding** *set\_Cons\_def* **by** *blast*

**lemma** *mergeable\_envs\_set\_Cons*: *mergeable\_envs n S*  $\implies \text{mergeable\_envs } (\text{Suc } n) (\text{set\_Cons UNIV } S)$   
**unfolding** *mergeable\_envs\_def*  
**proof** (*clarify, elim in\_set\_ConsE, goal\_cases mergeable*)  
**case** (*mergeable v1 v2 A B f y1 ys1 y2 ys2*)  
**let** *?A* = ( $\lambda x. x - 1$ ) ' (*A - {0}*)  
**let** *?B* = ( $\lambda x. x - 1$ ) ' (*B - {0}*)  
**from** *mergeable*(4-9) **have**  $\exists v \in S. \forall x \in ?A \cup ?B. v \ ! \ x = f (\text{Suc } x)$   
**by** (*auto dest!: mergeable*(2,3)[*rule\_format*] *intro!: mergeable*(1)[*rule\_format, of ys1 ys2*])  
**then obtain** *v* **where**  $v \in S \ \forall x \in ?A \cup ?B. v \ ! \ x = f (\text{Suc } x)$  **by** *blast*  
**then show** *?case*  
**by** (*intro beXI[of \_ f 0 # v]*) (*auto simp: nth\_Cons' set\_Cons\_def*)  
**qed**

**lemma** *slice\_Exists*: *MFOTL\_slicer.slice* (*MFOTL.Exists*  $\varphi$ ) *S*  $\sigma = \text{MFOTL\_slicer.slice } \varphi (\text{set\_Cons UNIV } S) \ \sigma$   
**by** (*auto simp: set\_Cons\_def intro: map\_Γ\_cong*)

**lemma** *image\_Suc\_fvi*: *Suc* ' *MFOTL.fvi* (*Suc b*)  $\varphi = \text{MFOTL.fvi } b \ \varphi - \{0\}$   
**by** (*auto simp: image\_def Bex\_def MFOTL.fvi\_Suc dest: gr0\_implies\_Suc*)

**lemma** *nfv\_Exists*: *MFOTL.nfv* (*MFOTL.Exists*  $\varphi$ ) = *MFOTL.nfv*  $\varphi - 1$   
**unfolding** *MFOTL.nfv\_def*  
**by** (*cases fv*  $\varphi = \{\}$ ) (*auto simp add: image\_Suc\_fvi mono\_Max\_commute[symmetric] mono\_def*)

```

lemma set_Cons_empty_iff[simp]: set_Cons A Xs = {}  $\longleftrightarrow$  A = {}  $\vee$  Xs = {}
  unfolding set_Cons_def by auto

lemma unique_sat_slice_mem: safe_formula  $\varphi \implies$  gen_unique  $\varphi \implies S \neq \{\} \implies$ 
  mergeable_envs n S  $\implies$  MFOTL.nfv  $\varphi \leq n \implies$ 
  MFOTL.sat (MFOTL.slicer.slice  $\varphi$  S  $\sigma$ ) v i  $\varphi \implies \exists v' \in S. \forall x \in fv \varphi. v' ! x = v ! x$ 
proof (induction arbitrary: v i S n rule: safe_formula_induct)
  case (1 t1 t2)
  then show ?case by (cases t2) (auto simp: MFOTL.is_Const_def)
next
  case (2 t1 t2)
  then show ?case by (cases t1) (auto simp: MFOTL.is_Const_def)
next
  case (3 x y)
  then show ?case by auto
next
  case (4 x y)
  then show ?case by simp
next
  case (5 e ts)
  then obtain v' where v'  $\in S$  and eq:  $\forall t \in set \ ts. MFOTL.eval\_trm \ v' \ t = MFOTL.eval\_trm \ v \ t$ 
  by auto
  have  $\forall t \in set \ ts. \forall x \in fv\_trm \ t. v' ! x = v ! x$  proof
    fix t assume t  $\in set \ ts$ 
    with eq have MFOTL.eval_trm v' t = MFOTL.eval_trm v t ..
    then show  $\forall x \in fv\_trm \ t. v' ! x = v ! x$  by (cases t) (simp_all)
  qed
  with  $\langle v' \in S \rangle$  show ?case by auto
next
  case (6  $\varphi \ \psi$ )
  from  $\langle gen\_unique \ (MFOTL.And \ \varphi \ \psi) \rangle$ 
  have
    MFOTL.sat (MFOTL.slicer.slice (MFOTL.And  $\varphi \ \psi$ ) S  $\sigma$ ) v i  $\varphi = MFOTL.sat (MFOTL.slicer.slice
 $\varphi$  S  $\sigma$ ) v i  $\varphi$ 
    MFOTL.sat (MFOTL.slicer.slice (MFOTL.And  $\varphi \ \psi$ ) S  $\sigma$ ) v i  $\psi = MFOTL.sat (MFOTL.slicer.slice
 $\psi$  S  $\sigma$ ) v i  $\psi$ 
    unfolding MFOTL.And_def
    by (fastforce simp: unique_names_matches_absorb_intro!: sat_inter_names_cong) +
  with 6(1,4-) 6(2,3)[where S=S] show ?case
    unfolding MFOTL.And_def
    by (auto intro!: mergeable_envs_Ex)
next
  case (7  $\varphi \ \psi$ )
  from  $\langle gen\_unique \ (MFOTL.And\_Not \ \varphi \ \psi) \rangle$ 
  have MFOTL.sat (MFOTL.slicer.slice (MFOTL.And\_Not  $\varphi \ \psi$ ) S  $\sigma$ ) v i  $\varphi = MFOTL.sat (MFOTL.slicer.slice
 $\varphi$  S  $\sigma$ ) v i  $\varphi$ 
    unfolding MFOTL.And\_Not_def
    by (fastforce simp: unique_names_matches_absorb_intro!: sat_inter_names_cong)
  with 7(1,2,5-) 7(3)[where S=S] have  $\exists v' \in S. \forall x \in fv \ \varphi. v' ! x = v ! x$ 
    unfolding MFOTL.And\_Not_def by auto
  with  $\langle fv \ \psi \subseteq fv \ \varphi \rangle$  show ?case by (auto simp: MFOTL.fvi_And_Not)
next
  case (8  $\varphi \ \psi$ )
  from  $\langle gen\_unique \ (MFOTL.Or \ \varphi \ \psi) \rangle$ 
  have
    MFOTL.sat (MFOTL.slicer.slice (MFOTL.Or  $\varphi \ \psi$ ) S  $\sigma$ ) v i  $\varphi = MFOTL.sat (MFOTL.slicer.slice
 $\varphi$  S  $\sigma$ ) v i  $\varphi$ 
    MFOTL.sat (MFOTL.slicer.slice (MFOTL.Or  $\varphi \ \psi$ ) S  $\sigma$ ) v i  $\psi = MFOTL.sat (MFOTL.slicer.slice$$$$$ 
```

```

ψ S σ) v i ψ
  by (fastforce simp: unique_names_matches_absorb intro!: sat_inter_names_cong)+
with 8(1,4-) 8(2,3)[where S=S] have ∃ v'∈S. ∀ x∈fv φ. v' ! x = v ! x
  by (auto simp: ⟨fv ψ = fv φ⟩)
then show ?case by (auto simp: ⟨fv ψ = fv φ⟩)
next
case (9 φ)
then obtain z where sat_φ: MFOTL.sat (MFOTL_slicer.slice (MFOTL.Exists φ) S σ) (z # v) i φ
  by auto
from 9.prem1 sat_φ have ∃ v'∈set_Cons UNIV S. ∀ x∈fv φ. v' ! x = (z # v) ! x
  unfolding slice_Exists
  by (intro 9.IH) (auto simp: nfv_Exists intro!: mergeable_envs_set_Cons)
then show ?case
  by (auto simp: set_Cons_def fvi_Suc Ball_def nth_Cons split: nat.splits)
next
case (10 I φ)
then obtain j where MFOTL.sat (MFOTL_slicer.slice φ S σ) v j φ
  by (auto split: nat.splits)
with 10 show ?case by simp
next
case (11 I φ)
then obtain j where MFOTL.sat (MFOTL_slicer.slice φ S σ) v j φ
  by (auto split: nat.splits)
with 11 show ?case by simp
next
case (12 φ I ψ)
from ⟨gen_unique (MFOTL.Since φ I ψ)⟩
have *:
  MFOTL.sat (MFOTL_slicer.slice (MFOTL.Since φ I ψ) S σ) v j ψ = MFOTL.sat (MFOTL_slicer.slice
ψ S σ) v j ψ for j
  by (fastforce simp: unique_names_matches_absorb intro!: sat_inter_names_cong)
from 12 obtain j where MFOTL.sat (MFOTL_slicer.slice (MFOTL.Since φ I ψ) S σ) v j ψ
  by auto
with 12 have ∃ v'∈S. ∀ x∈fv ψ. v' ! x = v ! x using * by auto
with ⟨fv φ ⊆ fv ψ⟩ show ?case by auto
next
case (13 φ I ψ)
from ⟨gen_unique (MFOTL.Since (MFOTL.Neg φ) I ψ)⟩
have *:
  MFOTL.sat (MFOTL_slicer.slice (MFOTL.Since (MFOTL.Neg φ) I ψ) S σ) v j ψ = MFOTL.sat
(MFOTL_slicer.slice ψ S σ) v j ψ for j
  by (fastforce simp: unique_names_matches_absorb intro!: sat_inter_names_cong)
from 13 obtain j where MFOTL.sat (MFOTL_slicer.slice (MFOTL.Since (MFOTL.Neg φ) I ψ) S
σ) v j ψ
  by auto
with 13 have ∃ v'∈S. ∀ x∈fv ψ. v' ! x = v ! x using * by auto
with ⟨fv (MFOTL.Neg φ) ⊆ fv ψ⟩ show ?case by auto
next
case (14 φ I ψ)
from ⟨gen_unique (MFOTL.Until φ I ψ)⟩
have *:
  MFOTL.sat (MFOTL_slicer.slice (MFOTL.Until φ I ψ) S σ) v j ψ = MFOTL.sat (MFOTL_slicer.slice
ψ S σ) v j ψ for j
  by (fastforce simp: unique_names_matches_absorb intro!: sat_inter_names_cong)
from 14 obtain j where MFOTL.sat (MFOTL_slicer.slice (MFOTL.Until φ I ψ) S σ) v j ψ
  by auto
with 14 have ∃ v'∈S. ∀ x∈fv ψ. v' ! x = v ! x using * by auto
with ⟨fv φ ⊆ fv ψ⟩ show ?case by auto

```

```

next
  case (15  $\varphi$  I  $\psi$ )
  from  $\langle \text{gen\_unique } (MFOTL.Until (MFOTL.Neg \varphi) I \psi) \rangle$ 
  have *:
    MFOTL.sat (MFOTL_slicer.slice (MFOTL.Until (MFOTL.Neg  $\varphi$ ) I  $\psi$ ) S  $\sigma$ ) v j  $\psi$  = MFOTL.sat
    (MFOTL_slicer.slice  $\psi$  S  $\sigma$ ) v j  $\psi$  for j
    by (fastforce simp: unique_names_matches_absorb intro!: sat_inter_names_cong)
  from 15 obtain j where MFOTL.sat (MFOTL_slicer.slice (MFOTL.Until (MFOTL.Neg  $\varphi$ ) I  $\psi$ ) S
   $\sigma$ ) v j  $\psi$ 
  by auto
  with 15 have  $\exists v' \in S. \forall x \in fv \psi. v' ! x = v ! x$  using * by auto
  with  $\langle fv (MFOTL.Neg \varphi) \subseteq fv \psi \rangle$  show ?case by auto
qed

```

```

lemma unique_sat_slice:
  assumes formula: safe_formula  $\varphi$  gen_unique  $\varphi$ 
  and restr:  $S \neq \{\}$  mergeable_envs (MFOTL.nfv  $\varphi$ ) S
  and sat_slice: MFOTL.sat (MFOTL_slicer.slice  $\varphi$  S  $\sigma$ ) v i  $\varphi$ 
  shows MFOTL.sat  $\sigma$  v i  $\varphi$ 
proof -
  obtain v' where v'  $\in$  S and fv_eq:  $\forall x \in fv \varphi. v' ! x = v ! x$ 
  using unique_sat_slice_mem[OF formula restr order_refl sat_slice] ..
  with sat_slice have MFOTL.sat (MFOTL_slicer.slice  $\varphi$  S  $\sigma$ ) v' i  $\varphi$ 
  by (auto iff: sat_fvi_cong)
  then have MFOTL.sat  $\sigma$  v' i  $\varphi$ 
  unfolding sat_slice_iff[OF  $\langle v' \in S \rangle$ , symmetric] .
  with fv_eq show ?thesis by (auto iff: sat_fvi_cong)
qed

```

### 7.2.8 Lemma 3

```

lemma (in splitting_strategy) unique_sat_strategy:
  safe_formula  $\varphi$   $\implies$  gen_unique  $\varphi$   $\implies$  slice_set k  $\neq \{\}$   $\implies$ 
  mergeable_envs (MFOTL.nfv  $\varphi$ ) (slice_set k)  $\implies$ 
  MFOTL.sat (MFOTL_slicer.slice  $\varphi$  (slice_set k)  $\sigma$ ) (map the v) i  $\varphi$   $\implies$ 
  ok  $\varphi$  v  $\implies$  k  $\in$  strategy v
  by (drule (3) unique_sat_slice_mem) (auto dest: wf_tuple_cong)

```

```

locale skip_inter = joint_data_slicer +
  assumes nonempty: slice_set k  $\neq \{\}$ 
  and mergeable: mergeable_envs (MFOTL.nfv  $\varphi$ ) (slice_set k)
begin

```

### 7.2.9 Definition of J'

```

definition skip_joiner = ( $\lambda s. \bigcup k. s k$ )

```

### 7.2.10 Theorem 3

```

lemma skip_joiner:
  assumes safe_formula  $\varphi$  gen_unique  $\varphi$ 
  shows joiner ( $\lambda k. M$  (splitter  $\pi$  k)) = skip_joiner ( $\lambda k. M$  (splitter  $\pi$  k))
  (is ?L = ?R)
proof safe
  fix i v
  assume (i, v)  $\in$  ?R
  then obtain k where in_M: (i, v)  $\in$  M (splitter  $\pi$  k)
  unfolding skip_joiner_def by blast
  from ex_prefix_of obtain  $\sigma$  where prefix_of  $\pi$   $\sigma$  by blast

```

```

with wf_monitor[OF in_M] sound_monitor[OF in_M] have
  MFOTL.sat (MFOTL slicer.slice  $\varphi$  (slice_set k)  $\sigma$ ) (map the v) i  $\varphi$  ok  $\varphi$  v
by (auto simp: splitter_pslice intro!: prefix_of_pmap_ $\Gamma$ )
note unique_sat_strategy[OF assms nonempty mergeable this]
with in_M show  $(i, v) \in ?L$  unfolding joiner_def by blast
qed (auto simp: joiner_def skip_joiner_def)

```

```

sublocale skip_joint_monitor: MFOTL_monitor  $\varphi$ 
   $\lambda\pi.$  (if safe_formula  $\varphi \wedge$  gen_unique  $\varphi$  then skip_joiner else joiner) ( $\lambda k. M$  (splitter  $\pi$  k))
using joint_monitor.mono_monitor joint_monitor.wf_monitor joint_monitor.sound_monitor joint_monitor.complete_mo
by unfold_locales (auto simp: skip_joiner[symmetric] split: if_splits)

```

**end**

## References

- [1] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.
- [2] D. Basin, F. Klaedtke, and E. Zălinescu. The MonPoly monitoring tool. In G. Reger and K. Havelund, editors, *RV-CuBES 2017*, volume 3 of *Kalpa Publications in Computing*, pages 19–28. EasyChair, 2017.
- [3] J. Schneider, D. Basin, F. Brix, S. Krstić, and D. Traytel. Scalable online first-order monitoring. *Int. J. Softw. Tools Technol. Transf.*, 2020. To appear. Preprint at [http://people.inf.ethz.ch/traytel/papers/sttt20-som\\_long/som\\_long.pdf](http://people.inf.ethz.ch/traytel/papers/sttt20-som_long/som_long.pdf).
- [4] J. Schneider, D. Basin, S. Krstić, and D. Traytel. A formally verified monitor for metric first-order temporal logic. In B. Finkbeiner and L. Mariani, editors, *RV 2019*, volume 11757 of *LNCS*, pages 310–328. Springer, 2019.